Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology

Programmer's Guide

August 2009

Order Number: 320183-004US

Contents

Figures

Tables

Revision History

Date	Revision	Description
August 2009	004	Added Note explaining cryptographic framework “shim” support (identified in “What’s New” sections of chapters and with change bars).
May 2009	003	The following sections were updated and noted with change bars: • OCF shim is now supported on FreeBSD; modified Notes in Section 1.6, Section 3.3.5, Section 12.9, and Section 12.10. • Modified Table 7, “Resource Variables” on page 33. • In Chapter 11.0, “Debugging Applications,” added Section 11.2.1 and Section 11.3.8. • Deleted Resource Manager text from Section 3.6 and Section 8.3.1 (no change bars). • Removed Software Error Notification section in Chapter 6 (no change bars). • Other updates noted in “What’s New” sections of chapters and with change bars.
November 2008	002	The following sections were updated and noted with change bars: • Section 12.8.2.6: added details on key generation • Other updates noted in “What’s New” sections of chapters and with change bars.
August 2008	001	Initial release of this document.

§ §

1.0 Introduction

1.1 What's New in this Chapter

Section 1.6: New Note explaining cryptographic framework "shim" support.

1.2 About this Document

The API Reference Manuals listed in Table 1 describe how the user can interface to the Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology. This document provides more information on how the APIs can be effectively used, including an overview of the silicon, an overview of the software architecture, and information on using the API to build an accelerated security appliance.

The following chapters are included in this document:

- Chapter 1.0, "Introduction" this chapter
- Part 1: "Architectural Overview"
  - — Chapter 2.0, "Silicon Overview"
  - — Chapter 3.0, "Software Overview"
  - — Chapter 4.0, "Intel® QuickAssist Technology Cryptographic API Architecture Overview"
  - — Chapter 5.0, "QAT Access Layer Architecture Overview"
  - — Chapter 6.0, "Debug Component Architecture Overview"
  - — Chapter 7.0, "ASD Module Architecture Overview"
  - — Chapter 8.0, "ASD Hardware Services"
- Part 2: "Using the API"
  - — Chapter 9.0, "Introduction to Use Cases"
  - — Chapter 10.0, "Programming Model"
  - — Chapter 11.0, "Debugging Applications"
  - — Chapter 12.0, "Using the Intel® QuickAssist Technology Cryptographic API"
  - — Appendix A, "NPF Copyright Notice"

1.3 Where to Find Current Software and Documentation

The software release and associated collateral can be found on the Hardware Design resource center.

1. In a web browser, go to http://www.intel.com/go/soc
2. For Software and pre-boot firmware: Click on "Tools & Software" tab.
3. For Documentation: Click on "Technical Documents" tab.

1.4 Related Information

Note: For convenience, in this document [GET_STARTED_GD] refers to either the Linux or FreeBSD guide. Refer to the appropriate guide for your operating system.

Table 1. Related Documents and Sample Code

Ref	Document Name	Document Number
[CRYPTO_API]	Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology Cryptographic API Reference Manual	320184
[GET_STARTED_GD]	Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology for Linux* Getting Started Guide Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology for FreeBSD* Getting Started Guide	320182 320703
[DEBUG_API]	Intel® EP80579 Software on Intel® QuickAssist Technology Debug Services API Reference Manual	320185
[SAMPLE_CODE]	After installation, sample code may be found here: Acceleration/library/icp_crypto/look_aside_crypto/sample_code/	N/A
[OCF_CODE]	After installation, OCF Shim sample code may be found here: Acceleration/shims/OCF_Shim/src/	N/A

1.4.1 Reference Documents

The following documents provide more information on certain topics beyond the scope of this guide.

Table 2. Reference Documents

Reference	Document Information
[4+1]	"The 4+1 View Model of Architecture" by Philippe B. Kruchten, Rational Software, Nov 1995, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=469759
[NPF API]	"NPF Software API Conventions Implementation Agreement", Revision 2.0, http://www.oiforum.com/public/documents/APIConventions2_IA.pdf ***Note:*** See Appendix A, "NPF Copyright Notice" for more information.
[IA-32]	Intel® 64 and IA-32 Architectures Software Developer's Manuals http://www.intel.com/products/processor/manuals/

1.5 Glossary

Table 3 lists the terms and acronyms used in this document.

Table 3. Terms and Definitions (Sheet 1 of 2)

Term	Description
(A)RC4	Alleged RC4. A stream cipher, used in popular protocols such as SSL
ACPI	Advanced Configuration and Power Interface
AES	Advanced Encryption Standard
AIOC	Acceleration and I/O Complex
APM	Advanced Power Management
ASD	Acceleration System Driver
ASU	Acceleration Services Unit

Table 3. Terms and Definitions (Sheet 2 of 2)

Term	Description
CBC	Cipher Block Chaining mode. This is a mode of operation of a block cipher that combines the ciphertext of one block with the plaintext of the next block.
CDRAM	Coherent DRAM
CTR	Counter mode. This is a mode of operation of a block cipher that generates a keystream block by encrypting successive values of a counter.
DES	Data Encryption Standard
DRAM	Dynamic Random Access Memory
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECB	Electronic Code Book
GbE	Gigabit Ethernet
GCM	Galois Counter Mode
HMAC	Hashed Message Authenticate Code
IICH	Integrated I/O Controller Hub
IMCH	Integrated Memory Controller Hub
IPSec	Internet Protocol Security
IV	Initialization Vector
LA	Lookaside, also called Cryptographic API
LAC	Lookaside Crypto, also called Cryptographic API
MAC	Media Access Control
MD5	Message Digest 5
MGF	Mask Generation Function
NCDRAM	Non-Coherent DRAM
NVRAM	Non-Volatile Random Access memory
OCF	OpenBSD Cryptographic Framework
OIF	Optical Internetworking Forum, standards body for networking specifications
OSDRAM	Operating System DRAM
PKCS	Public Key Cryptography Standards
PKE	Public Key Encryption
PKI	Public Key Infrastructure
PRGA	Pseudo-Random Generation Algorithm
QAT-AL	Intel® QuickAssist Technology Access Layer
RC4	See (A)RC4
RFC	Request for Comments
RSA	A public key encryption algorithm created by **R**ivest, **S**hamir, and **A**dleman
SHA	Secure Hash Algorithm
SOC	System on a chip
SSL	Secure Sockets Layer
SSU	Security Services Unit
TDM	Time-Division Multiplexing
TLS	Transport Layer Security (SSL successor)

1.6 Features Supported in this Release

The features provided by this software in this release are as follows:

- Acceleration of cryptographic operations using the "lookaside" model, via the Cryptographic API. For more details, see Chapter 4.0, "Intel® QuickAssist Technology Cryptographic API Architecture Overview."
  - — Symmetric cryptographic operations supported include ciphers [AES, 3DES, DES, (A)RC4] and message digest/hash for authentication (MD5, SHA-1, SHA-2 as well as HMAC).
  - — Asymmetric (public key) cryptographic operations such as modular exponentiation to support RSA, Diffie-Hellman, DSA.
  - — "True" random number generation.

  This allows for cryptographic protocols such as IPSec and SSL to offload compute-intensive cryptographic operations, freeing up the IA core to execute higher-value application code.
- Software is provided which adapts between the Cryptographic API and that expected by the industry-standard OpenBSD Cryptographic Framework (OCF). This OCF "shim" allows applications — such as Openswan* and OpenSSL* —which are written to use the OCF APIs, to seamlessly take advantage of the cryptographic acceleration engine.

  Note: The EP80579 security software release package version 1.0.3 does **not** support OpenBSD/FreeBSD Cryptographic Framework (OCF), OCF-Linux, or any open source projects such as Openswan*, OpenSSL*, or Racoon*. If your application requires OCF, you must use security software package version 1.0.2 which includes shim software to enable OCF support.

§ §

Part 1: Architectural Overview

This section contains the following chapters:

§ §

2.0 Silicon Overview

2.1 What's New in this Chapter

No updates in this release.

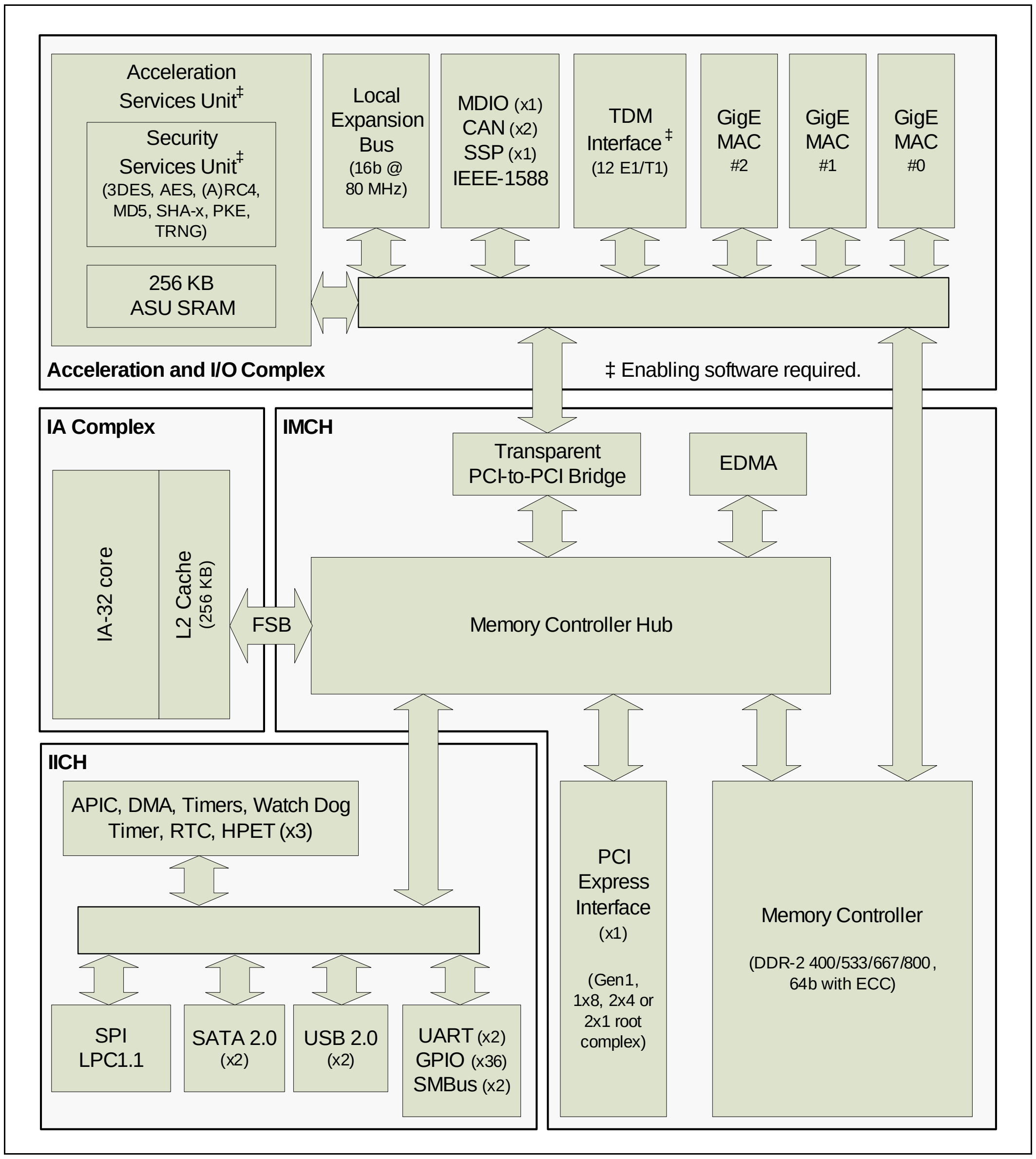
2.2 High Level Overview

The Intel® EP80579 Integrated Processor is a System On a Chip (SOC), integrating the Intel® Architecture core processor, the Integrated Memory Controller Hub (IMCH) and the Integrated I/O Controller Hub (IICH) all on the same die. In addition, it has integrated Intel® QuickAssist Technology, which provides acceleration of cryptographic and packet processing. It also includes three gigabit Ethernet MACs, TDM interfaces, and PCI Express. See Figure 1 for details.

- As an SOC, the EP80579 integrates the processor and chipset as follows:
  - — The IA-32 core is based on the Intel® Pentium® M processor, and runs at 600-1200MHz, with a 256 Kilobyte 2-way level 2 (L2) cache.
  - — The IMCH provides the main path to memory for the IA core and all peripherals that perform coherent I/O (for example, the PCI express, the IICH, as well as transactions from the Acceleration and I/O Complex to coherent memory).
  - — The IICH provides a set of PC platform-compatible I/O devices that include two SATA 1.0/2.0, two USB 1.1/2.0 host controller supporting two USB ports, and two serial 16550 compatible UART interfaces.
- The Intel® QuickAssist Technology components, housed in the Acceleration and I/O Complex (AIOC), are as follows:
  - — The Security Services Unit (SSU) provides acceleration of cryptographic processing for most common symmetric cryptography (cipher algorithms such as AES, 3DES, DES, (A)RC4, and messages digest/hash functions such as MD5, SHA-1, SHA-2, HMAC, etc.); asymmetric cryptography (modular exponentiation to support public key encryption such as RSA, Diffie-Hellman, DSA); and true random number generation.
  - — The Acceleration Services Unit (ASU) includes packet processing acceleration engines.
- Other components within the AIOC include:
  - — Three Gigabit Ethernet (GbE) media access controllers (MACs).
  - — Three High Speed Serial (HSS) interfaces that support up to 12 T1/E1 TDM interfaces. These interfaces are driven by a *Programmable I/O Unit (PIU)*. The PIU is not part of the ASU. In Figure 1 on page 13, the PIU is shown as the TDM Interface block.
  - — Although not shown explicitly in Figure 1, the AIOC also contains logic to allow agents to access on-chip SRAM and external DRAM. Based on registers which can be configured in the BIOS, this logic routes requests to external DRAM either directly to the memory controller (to access non-coherent DRAM, or NCDRAM); or through the IMCH for coherency with the IA processor's L2 cache (to access Coherent DRAM, or CDRAM). There is also a ring controller, which

provides 64 rings (circular buffers) that can be used for message passing between software running on the IA core and firmware running on the ASU. These features are described in detail in later sections of this document.

Figure 1. Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology Block Diagram


Acceleration Services Unit‡
Security Services Unit‡
(3DES, AES, (A)RC4, MD5, SHA-x, PKE, TRNG)
256 KB ASU SRAM
Local Expansion Bus (16b @ 80 MHz)
MDIO (x1) CAN (x2) SSP (x1) IEEE-1588
TDM Interface‡ (12 E1/T1)
GigE MAC #2
GigE MAC #1
GigE MAC #0
Acceleration and I/O Complex
‡ Enabling software required.
IA Complex
IA-32 core
L2 Cache (256 KB)
FSB
IMCH
Transparent PCI-to-PCI Bridge
EDMA
Memory Controller Hub
IICH
APIC, DMA, Timers, Watch Dog Timer, RTC, HPET (x3)
SPI LPC1.1
SATA 2.0 (x2)
USB 2.0 (x2)
UART (x2) GPIO (x36) SMBus (x2)
PCI Express Interface (x1) (Gen1, 1x8, 2x4 or 2x1 root complex)
Memory Controller (DDR-2 400/533/667/800, 64b with ECC)

§ §

3.0 Software Overview

This chapter presents the high-level architecture of the Software for Intel® EP80579 Integrated Processor product line, using concepts from the "4+1 view model" of software architecture, as described in [4+1]. These views are interpreted as follows:

- Section 3.3, "Logical View" on page 15 describes the collection of software components in terms of their key responsibilities, interfaces, and dependencies.
- Section 3.4, "Development View" on page 17 describes the static organization of the software in its development environment (that is, folders and files).
- Section 3.5, "Process View" on page 18 captures concurrency and synchronization aspects of the architecture. This includes the mapping of software onto hardware, reflecting the distributed aspect of the architecture; this is sometimes considered part of the Physical or Deployment View.
- Section 3.6, "Deployment View" on page 18 describes the mapping of the software into kernel modules.
- The architecture is illustrated with a few selected use cases or scenarios which become a fifth view, the Scenario View. In this document, the Scenario View is described in Part 2, "Using the API" on page 41.

Before looking at these views, however, other concepts relevant to the architecture are introduced:

- Section 3.2, "Shared Memory Allocation" on page 14 describes the concepts of coherent and non-coherent DRAM.

3.1 What's New in this Chapter

- Section 3.3.5: New Note explaining cryptographic framework "shim" support.

3.2 Shared Memory Allocation

Two regions of memory exist outside of the normal operating system DRAM, to facilitate communications between the IA core and the EP80579 with QuickAssist hardware. These are referred to as the coherent and non-coherent shared memory regions.

These shared memory regions will be allocated from the available system memory, starting at the address specified by the Top Of Low Memory (TOLM) register of the Memory Controller Hub (MCH) downwards. The pre-boot firmware (BIOS) informs the operating system of the location of the regions, and also configures the hardware to properly decode the non-coherent memory space by writing the MENCBASE and MENCLIMIT registers.

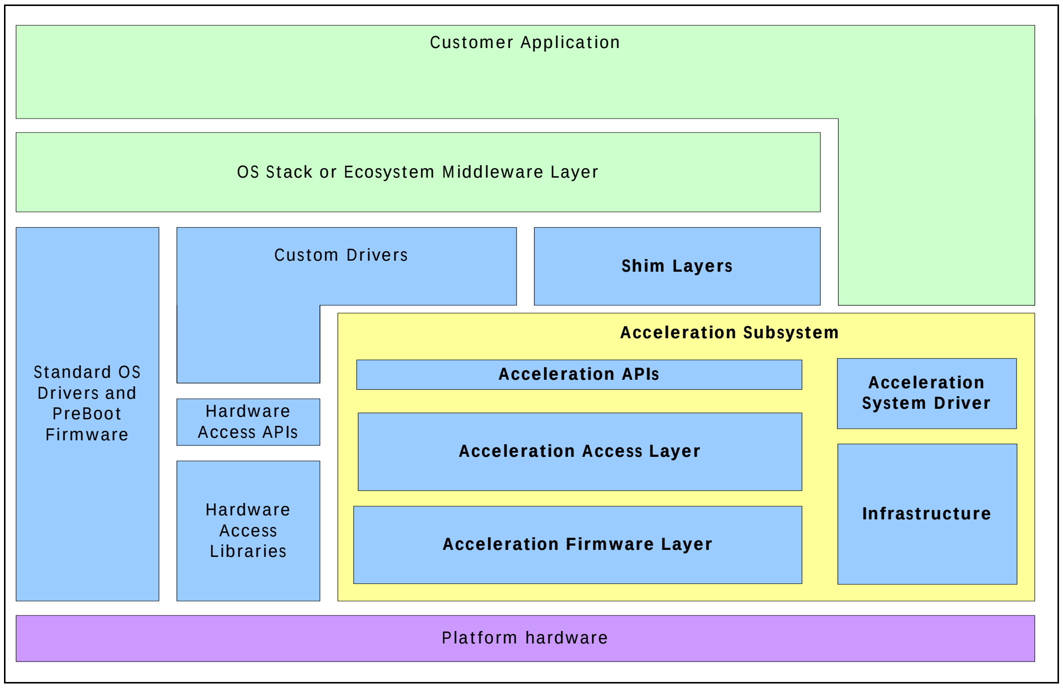
The base addresses for each of these regions will be determined by the firmware based on available memory. Two EFI NVRAM (non-volatile RAM) variables are available for the user to request a specific amount of space for each of these shared memory regions. The firmware will make a best effort to accommodate the user's request, but in the event this is not possible, the firmware will determine the sizes of these regions and set them accordingly. See Chapter 8.0, "ASD Hardware Services" for details on how this is configured.

3.3 Logical View

At the highest level, the software components fall into the following "layers", as illustrated in Figure 2.

In this document, and for this release, only those layers highlighted in **bold** are described in more detail.

Figure 2. Software for Intel® EP80579 Integrated Processor product line


Customer Application
OS Stack or Ecosystem Middleware Layer
Custom Drivers
Shim Layers
Standard OS Drivers and PreBoot Firmware
Acceleration Subsystem
Acceleration APIs
Acceleration System Driver
Hardware Access APIs
Acceleration Access Layer
Infrastructure
Hardware Access Libraries
Acceleration Firmware Layer
Platform hardware

3.3.1 Acceleration Firmware Layer

This layer of the architecture is for firmware which runs on the ASU.

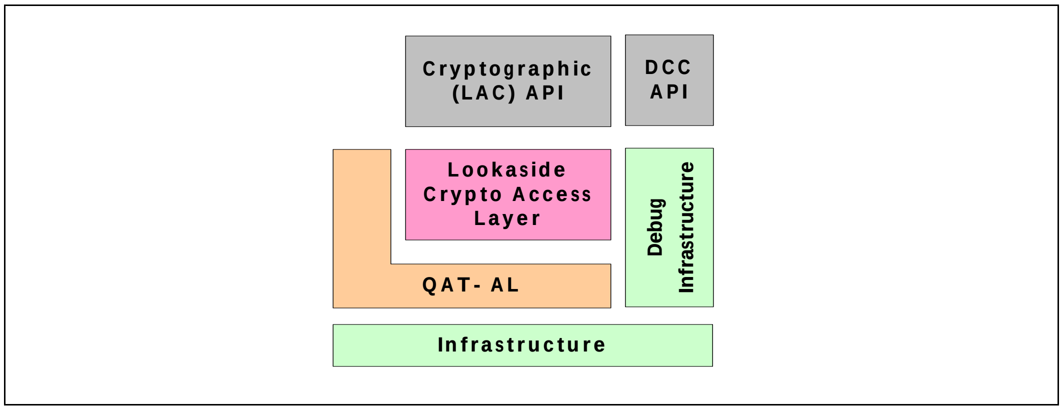
The only firmware running at this layer in this software release is the firmware driver for the SSU, which runs on the ASU. This firmware is provided in binary format.

3.3.2 Acceleration Access Layer and Acceleration APIs

This layer of software runs on the IA core. It implements the configuration and control of the Acceleration Firmware layer running on the ASU, and provides an Application Programming Interface (API) for the rest of the system to interface with the acceleration firmware.

Figure 3 shows the different components at this layer. The APIs are also shown to highlight the mapping between APIs and the corresponding acceleration libraries.

Figure 3. Acceleration Access Layer and Acceleration APIs



The software components at this layer in the current release are as follows:

- Lookaside Crypto Access Layer: This component implements the Cryptographic API (shown as LAC API in Figure 3). It manages the exchange of data and messages between the Cryptographic API and the SSU driver firmware running on the ASU. See Chapter 4.0, "Intel® QuickAssist Technology Cryptographic API Architecture Overview" for more details.
- QAT Access Layer: This component implements the configuration and control of the SSU driver firmware running on the ASU. It also provides an interface for the Lookaside Crypto Access Layer to communicate with the SSU driver firmware. See Chapter 5.0, "QAT Access Layer Architecture Overview" for more details.
- Debug Infrastructure: This component provides access to data which can be used to help debug an application running on EP80579 with QuickAssist. It allows version information to be queried, "liveness" of components to be polled, data dumps to be generated which can be analyzed offline, and other debug-related features. See Chapter 6.0, "Debug Component Architecture Overview" for more details.

  Note: The Data Dump feature is not supported in the current software release.

Most of the layers above also provide APIs. These are described in more detail in the chapters which comprise Part 2: "Using the API" on page 41.

3.3.3 Infrastructure

This layer consists of the following components:

- The Hardware Services Layer (HSL) component manages the low-level hardware blocks required for communication with the ASU. This also provides an interface for exchanging messages with the ASU via rings.
- The Operating System Abstraction Layer (OSAL) component provides OS-specific services. It is used by many of the components to remove their dependency on a particular OS and allow for easier porting to new OSes.

3.3.4 Acceleration System Driver (ASD)

The ASD is a system device driver which is responsible for loading firmware and configuring all the components that comprise the EP80579 security software. It initializes the Cryptographic API Library, providing it with all necessary information about the enumeration of the Acceleration Services Unit and any Access library specific configuration parameters for example number of sessions to be supported, buffer pool sizes, and so on. See Chapter 7.0, "ASD Module Architecture Overview" for more details.

3.3.5 Shim Layers

Note: The EP80579 security software release package version 1.0.3 does **not** support OpenBSD/FreeBSD Cryptographic Framework (OCF), OCF-Linux, or any open source projects such as Openswan*, OpenSSL*, or Racoon*. If your application requires OCF, you must use security software package version 1.0.2 which includes shim software to enable OCF support.

This layer is intended for components which adapt, or "shim", between the API provided by EP80579 security software's Acceleration API and that expected by industry-standard frameworks.

In this release, the only component in this layer is the OCF shim, which allows the lookaside crypto acceleration engine to be plugged in underneath the OpenBSD*/FreeBSD* Cryptographic Framework (OCF). OCF is a service virtualization layer that facilitates asynchronous access to cryptographic hardware accelerators. OCF-Linux is a port of this framework to Linux. It enables cryptographic acceleration in the Openswan* and OpenSSL* software suites.

A driver has been created which enables the Cryptographic API features to be accessed via OCF. All operations supported by OCF today are accelerated. Specifically, the following operations provided by OCF are accelerated by the OCF shim:

- Symmetric/Secret Key Crypto
  - — Ciphers/Modes: NULL_CBC, DES_CBC, 3DES_CBC, AES_CBC, ARC4
  - — Hash/Message Digest Functions: MD5, MD5_HMAC, SHA1, SHA1_HMAC, SHA2_256, SHA2_256_HMAC, SHA2_384, SHA2_384_HMAC, SHA2_512, SHA2_512_HMAC
  - — Chained Algorithms
- Asymmetric/Public Key Crypto
  - — Diffie-Hellman: DH_COMPUTE_KEY
  - — RSA: MOD_EXP, MOD_EXP_CRT
  - — DSA: DSA_SIGN, DSA_VERIFY
- Random Number Generation

See the [GET_STARTED_GD] for your operating system for detailed information.

Further information on OCF-Linux can be found here: http://ocf-linux.sourceforge.net

3.4 Development View

Table 4 describes the mapping between the software components described in Section 3.3, "Logical View" on page 15, and the files and directories (folders) in which they can be found.

Table 4. **Development View**

Software Component	Directory
OCF Shim	Acceleration/shims/OCF_Shim
Acceleration System Driver	Acceleration/drivers/icp_asd
Lookaside Crypto Access Layer	Acceleration/library/icp_crypto/look_aside_crypto
QAT Access Layer	Acceleration/library/icp_crypto/QATAL
Debug Infrastructure	Acceleration/library/icp_debug/DCC
Management Interface Module	Acceleration/library/icp_debug/MIL
Hardware Abstraction Layer (part of Hardware Services Layer)	Acceleration/library/icp_services/RuntimeTargetLibrary
Operating System Abstraction Layer	Acceleration/library/icp_utils/OSAL
Firmware Driver for SSU	Acceleration/firmware

3.5 Process View

This section describes the context in which the EP80579 security software code is executed, which is important in terms of understanding concurrency, or where locking may be required, for example.

Code which implements the Acceleration APIs is library code, and is executed in the context of whatever thread or interrupt context from which it is called. All of the EP80579 security software APIs document the context in which they can be called, specifically whether they may sleep and therefore are suitable for calling in a context which may not sleep, such as ISRs or certain types of "bottom halves" including softirq and tasklet. They also document whether they are thread-safe. Table 1, “Related Documents and Sample Code” on page 8 lists the API documentation supported in this release.

The remainder of EP80579 security software code runs in a well-defined context, whether it is process context or some form of interrupt context as described below.

- Interrupt handlers are registered for all interrupts from devices managed by EP80579 security software, specifically the GbE MACs, and the ring controller on the ASU. This code runs in the ISR (interrupt top half) context.
- Many of the Acceleration APIs support one or both of asynchronous and synchronous modes.
  - — In asynchronous mode: when the request has been carried out on the SSU, a “function completion callback” is typically invoked in a non-sleeping bottom half context (specifically, a tasklet, on Linux). For more on this topic, see Section 10.0, “Programming Model” on page 43.
  - — In synchronous mode: when the request has been sent to the SSU, the calling thread is blocked, pending on a wait queue. When the response is received from the SSU, the calling thread is de-queued, and thereby unbocked.

3.6 Deployment View

Table 5 describes the mapping between the software components described in Section 3.3, “Logical View” on page 15, and the kernel modules that are created by the build system.

Table 5. Deployment View

Kernel Module	Component
icp_asd.ko	Acceleration System Driver
icp_crypto.ko	Lookaside Crypto Access Layer QAT Access Layer
icp_debug.ko	Debug Infrastructure
icp_debugmgmt.ko	Management Interface Module ***Note:*** This is an optional kernel module, needed only if you are using the debugmgr command line utility described in Chapter 11.0, "Debugging Applications."
icp_hal.ko	Hardware Abstraction Layer
icp_ocf.ko	OCF Shim

§ §

4.0 Intel® QuickAssist Technology Cryptographic API Architecture Overview

4.1 What's New in this Chapter

No updates in this release.

4.2 Feature List

The Intel® QuickAssist Technology Cryptographic API comprises two broad feature areas in its API, they are the symmetric operations API and the public key cryptography API.

4.2.1 Symmetric Operations

4.2.1.1 Cipher

EP80579 security software supports the following Cipher algorithms:

- **AES** (128-bit/192-bit/256-bit key size) in ECB, CBC and CTR modes. Block size for data is 16 byte blocks.
- **3DES** (192-bit key size) in ECB and CBC and CTR mode. Block size for data is 8 bytes.
- **DES** (64-bit key size) in ECB and CBC mode. Block size for data is 8 bytes.
- **ARC4** (stream cipher)
- **NULL** cipher with a minimum block size of 8 bytes

4.2.1.2 Hash/Authentication

EP80579 security software supports the following Hash/Authentication algorithms:

- Secure Hash Algorithm SHA-1, SHA-224/256/384/512.
- Authentication algorithms for Secure Hash supported HMAC-SHA-1, HMAC-SHA-224/256/384/512
- Message Digest 5 (MD5) and HMAC-MD5
- Advanced Encryption Standard (AES) using 96-bit key in AES-XCBC mode to produce AES-XCBC-MAC-96.

4.2.1.3 Partial Packets for Cipher and Hash/Authentication Commands

A partial packet is defined as a portion of a full packet. The caller issues a separate request for each portion (partial packet) of the full packet. The size of data sent must be a multiple of the underlying algorithm block size for cipher and hash requests except for the final hash partial packet in which padding will be applied if it is not a block size. The final result following completion of all the portions is equivalent to the case where the operation is performed over the full packet in a single request. Partial-packet support is provided for Lookaside Cipher and Hash/Authentication commands only. Partial-packet support is not provided for any other commands.

The authentication result is not available until after the “final” operation has completed. The user provided callback will be called in all the cases.

From a user’s perspective, partial packets allow the client to send data to be processed when they receive it instead of buffering up an entire message. For example, consider the scenario where a digest needs to be created across gigabytes of data which is being accessed over a network interface. Rather than copying the entire data set to the platform, then performing a hash operation across all of the data, the client application could optimize this process by transferring blocks which are optimal for the network interface, then sending these chunks to the Lookaside security service for processing as they are received. This results in higher performance as the acceleration is being utilized while the transfers are being processed.

4.2.1.4 Out-Of-Place Operation Support

An Out-of-Place operation is when the result of a symmetric operation is written to the destination buffer. The destination buffer is a different physical location than the source buffer.

Note: In the current release, Out-of-Place operations are supported for full packets only.

4.2.1.5 Combined Cipher Hash Commands (Algorithm-Chaining)

Chained commands perform a cipher and a hash/authentication operation on the same input data. These commands are provided to allow more-optimal overall performance by minimizing the number of memory reads/writes for applications that require both cipher and hash/authentication operations on the same data. Only standard Cipher and Standard Hash/Authentication can be chained.

The algorithms mentioned in the Cipher and Hash/Authentication sections can be placed in any combination of one standard cipher and one standard hash / authenticate command. Combined Cipher and Hash Commands do not support partial packets.

When performing an authentication/hash prior to a cipher operation using the combined Cipher-Hash feature, the resultant MAC/digest produced by the authentication/hash cannot be included in the same cipher operation. The result of the authentication/hash operation will not be available for the cipher portion of the operation. This makes this feature unsuitable for SSL type authenticate-then-encrypt operations, where the MAC is included in the encryption.

4.2.1.6 Authenticated-Encryption Commands

Authenticated-Encryption commands perform chained cipher-and-authenticate operations. As in the case of other chained operations, these commands are provided to allow more-optimal overall performance by minimizing the number of memory reads/writes for applications that require both cipher and authentication operations on the same data.

The following Authenticated-Encryption algorithms are supported:

- AES algorithm in Galois/Counter mode (GCM)
- AES algorithm in Counter with CBC-MAC mode (CCM)

No partial packet support is provided for authentication encryption commands.

4.2.1.7 Key Generation

EP80579 security software supports the following Key Generation operations:

- SSL/TLS Key Generation
- MGF Mask Generation

4.2.2 Random Number

EP80579 security software supports the following Random Number operations:

- Random Data Generation
- Random Data Generator Seed (performed automatically by the hardware)

4.2.3 Public Key Operations

4.2.3.1 Diffie-Hellman

EP80579 security software supports the following Diffie-Hellman operations:

- Public/Private Key Generation (for Diffie-Hellman phase 1)
- Shared Secret Key Generation (for Diffie-Hellman phase 2)

4.2.3.2 RSA

EP80579 security software supports the following RSA operations:

- RSA Key Generation
- RSA Encryption/Decryption
- RSA Signature Generation/Verification

4.2.3.3 DSA

EP80579 security software supports the following DSA operations:

- DSA P, G and Y parameter generation.
- DSA Signature Generation/Verification

4.2.3.4 Prime Number

EP80579 security software supports the following prime number operations:

- Prime Number Tests (using GCD, Miller-Rabin, Lucas and Fermat)

4.2.3.5 Large Number

EP80579 security software supports the following large number operations:

- Modular Exponentiation
- Modular Inversion

4.3 Intel® QuickAssist Technology Cryptographic API Documentation

Refer to [CRYPTO_API] for more information about the Intel® QuickAssist Technology Cryptographic API.

4.4 Lookaside Security Algorithms High Level Overview

The following sections provide a high level overview of the algorithms supported by the Cryptographic API library. It details the algorithms and tries to pull out key details of the computations. For the reader who wants to get further details or specifics, it is recommended to reference the relevant RFC.

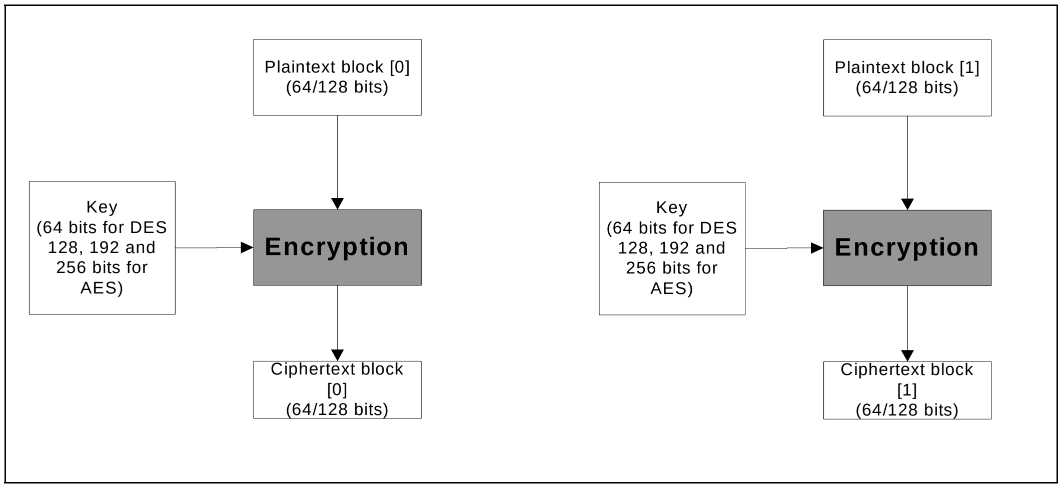
4.4.1 Lookaside Symmetric Overview

A **block cipher** is a symmetric key cipher that operates on fixed-length groups of bits, termed *blocks*, with an unvarying transformation. When encrypting, a block cipher might take a (for example) 128-bit block of plaintext as input, and output a corresponding 128-bit block of ciphertext. The exact transformation is controlled using a second input — the secret key. Decryption is similar; the decryption algorithm takes a 128-bit block of ciphertext together with the secret key, and yields the original 128-bit block of plaintext.

To encrypt messages longer than the block size (128 bits in the above example), a mode of operation is used.

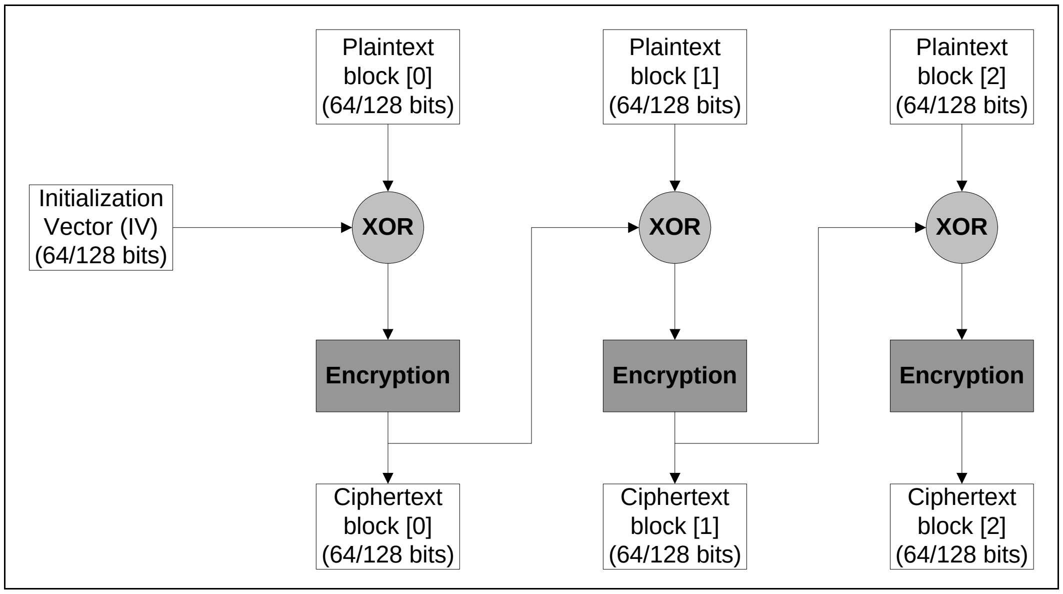
The simplest of the encryption modes is the **electronic codebook (ECB)** mode, in which the message is split into blocks and each is encrypted separately, as shown in Figure 4. The disadvantage of this method is that identical plaintext blocks are encrypted to identical cipher text blocks; it does not hide data patterns. Thus, in some senses it doesn't provide message confidentiality at all, and is not recommended for cryptographic protocols.

Figure 4. Electronic Codebook (ECB) Mode


Plaintext block [0] (64/128 bits)
Key (64 bits for DES 128, 192 and 256 bits for AES)
Encryption
Ciphertext block [0] (64/128 bits)
Plaintext block [1] (64/128 bits)
Key (64 bits for DES 128, 192 and 256 bits for AES)
Encryption
Ciphertext block [1] (64/128 bits)

In **cipher-block chaining (CBC)** mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted, as shown in Figure 5. This way, each ciphertext block is dependent on all plaintext blocks up to that point.

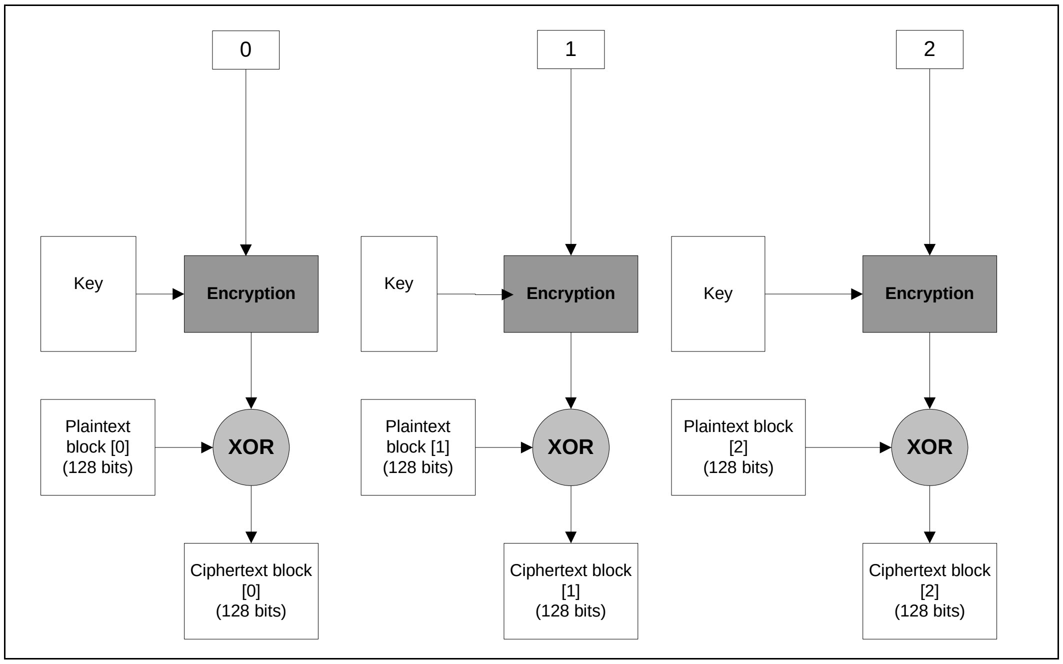
Figure 5. Cipher-Block Chaining (CBC) Mode


Plaintext block [0] (64/128 bits)
Plaintext block [1] (64/128 bits)
Plaintext block [2] (64/128 bits)
Initialization Vector (IV) (64/128 bits)
XOR
XOR
XOR
Encryption
Encryption
Encryption
Ciphertext block [0] (64/128 bits)
Ciphertext block [1] (64/128 bits)
Ciphertext block [2] (64/128 bits)

Note: Exclusive disjunction (usual symbol xor) is a logical operator that results in true if one of the operands (not both) is true.

Counter mode turns a block cipher into a stream cipher, as shown in Figure 6. It generates the next keystream block by encrypting successive values of a "counter". The counter can be any simple function which produces a sequence which is guaranteed not to repeat for a long time, although an actual counter is the simplest and most popular.

Figure 6. Counter Mode


0
1
2
Key
Encryption
Key
Encryption
Key
Encryption
Plaintext block [0] (128 bits)
XOR
Plaintext block [1] (128 bits)
XOR
Plaintext block [2] (128 bits)
XOR
Ciphertext block [0] (128 bits)
Ciphertext block [1] (128 bits)
Ciphertext block [2] (128 bits)

Note: A stream cipher operates on individual digits each one at a time.

(A)RC4

(A)RC4 generates a pseudorandom stream of bits (a “keystream”) which, for encryption, is combined with the plaintext using XOR a decryption is performed the same way. To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

- A permutation of all 256 possible bytes (denoted "S" below)
- Two 8-bit index-pointers (denoted "i" and "j")

The permutation is initialized with a variable length key, typically between 40 and 256 bits, using the key-scheduling algorithm (KSA). Once this has been completed, the stream of bits is generated using the Pseudo-Random Generation Algorithm (PRGA).

For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA increments *i*, adds the value of S pointed to by *i* to *j*, exchanges the values of S[*i*] and S[*j*], and then outputs the value of S at the location S[i] + S[j] (modulo 256). Each value of S is swapped at least once every 256 iterations.

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
```

NULL-ECB

The NULL cipher in ECB mode of operation simply produces the same plaintext as was passed into the algorithm.

Hashing / MAC / HMAC

A hash operation takes arbitrary binary data as input and produces a fixed-sized binary string as output called a hash or message digest. A cryptographic message authentication code (MAC) is a short piece of information used to authenticate a message. A MAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a MAC. The MAC value protects both a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. MAC functions are similar to keyed hash functions.

MAC algorithms can be constructed from other cryptographic primitives, such as cryptographic hash functions (as in the case of HMAC) or from block cipher algorithms (CBC-MAC and XCBC-MAC).

CCM

By definition, CCM is CTR Encryption and CBC-MAC Authentication. So AES-CCM is AES-CTR Encryption, AES-CBC-MAC Authentication. The valid key sizes for CTR mode are - 128/192/256 and for Authentication are 128 keys.

GCM

"Galois/Counter Mode (GCM) is a block cipher mode of operation that uses universal hashing over a binary Galois field to provide authenticated encryption." This is an excerpt from the GCM specification which can be accessed at: http://www.nist.gov/

4.4.2 Key Generation

The Cryptographic API module provides TLS and SSL key generation operation along with a Mask Generation Function (MGF).

TLS / SSL Generation: For both algorithms functions are provided for the generation of the Master-Secret and Key Materials. These are optimized accelerations for use in SSL/ TLS key negotiation and generation applications.

MGF: Takes a seed of specified length and produces a generated mask, which is pseudorandom, of the specified size.

4.4.3 Lookaside PKE Overview

This section gives a brief overview of Public Key algorithms and standards relevant for EP80579 security software. The following is a list of Public key algorithms/standards:

- Diffie-Hellman (DH) Key Exchange – PKCS #3 v1.4
- RSA Cryptography Standard – PKCS #1 v2.1 and ANSI X9.31
- Digital Signature Algorithm (DSA) – FIPS-186-2
- GCD, Miller-Rabin, Lucas and Fermat primality testing (ANSI X9.80)

4.4.3.1 Diffie-Hellman Key Exchange

DH is used to create a "shared secret", from which symmetric key information may be derived. This Key can be used to encrypt subsequent communications using a symmetric key cipher.

The protocol has two system parameters *p* and *g*. They are both public and may be used by all the users in a system. Parameter *p* is a prime number and parameter *g* (usually called a generator) is an integer less than *p*, with the following property: for every number *n* between 1 and *p*-1 inclusive, there is a power *k* of *g* such that $n = g^k$ mod p.

The underlying mathematical principle is the identity: $(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$. DH cryptographic strength is derived from the fact that logarithms are difficult to do in a MODP group. A set of standard DH (MODP) groups are defined in RFC-2409 and RFC-3526. Modulus sizes range from 768 to 4096 bits.

There are two modes of Diffie-Hellman:

- **Normal Diffie-Hellman**: DH parameters are contained within a certificate, signed by a certificate authority (CA).
- **Ephemeral Diffie-Hellman**: DH parameters are created "on the fly" by the negotiating parties. These parameters are then signed using a DSS or RSA certificate, which is itself signed by a CA.

4.4.3.2 RSA Cryptographic Standard

RSA may be used for encryption or signature generation. The Chinese Remainder Theorem (CRT) can be used as a method of RSA acceleration. CRT describes how to do exponentiation (or multiplication) modulo a composite modulus n as a series of smaller multiplications modulo the prime factors of n. Its cryptographic strength is derived from the fact that it is difficult to factor large composite numbers.

When used for encryption, the message is encapsulated using the PKCS v1.5 (deprecated) or OAEP (Optimal Asymmetric Encryption Padding) encoding schemes. OAEP is an improvement over the v1.5 (encryption) encoding scheme in that it provides security against adaptive chosen-ciphertext attacks.

When used for digital signatures, the message is encapsulated using the PKCS v1.5 (deprecated) or PSS (Probabilistic Signature Scheme). Although there are no known attacks against the PKCS v1.5 (signature) encoding scheme, the PSS encoding is more robust, as it introduces randomness into the encoded message, so that the same plaintext message will, in general, produce different encoded messages.

The above schemes (PKCS v1.5, OAEP, DSS, PSS) are supported by the Cryptographic API through supporting RSA primitive operations. There are no specific APIs to perform the encapsulation of the encryptions/signatures generated by the RSA primitive operations.

4.4.3.3 Digital Signature Algorithm

DSA is used for signature generation and verification only. It is a digital signature rather than a written signature. The DSA provides the capability to generate and verify signatures. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key which corresponds to, but is not the same as, the private key. Each user possesses a private and public key pair. Public keys are assumed to be known to the public in general. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key.

A hash function is used in the signature generation process to obtain a condensed version of data, called a message digest. The message digest is then input to DSA to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data. The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process.

The underlying mathematical principle is Fermat’s Little Theorem, which states that gp-1 mod p = 1 for p prime. Its cryptographic strength is derived from the fact that logarithms are difficult to do in a MODP group. As with Diffie-Hellman, DSA may be applied in an ephemeral manner, in which parameters are generated on the fly and used to create only one digital signature.

4.4.3.4 Prime Number Testing

Lookaside provides an interface to test probabilistically if a number is prime (refer to ANSI x9.80 specification for details). This is used for testing the primality of random numbers generated for key material. The following algorithms are supported for prime number sizes (in bits) 160, 512, 768, 1024, 1536, 2048, 3072 and 4096.

- GCD
- Fermat
- Miller-Rabin
- Lucas

Prime number testing can gain a performance improvement through parallelism of the requests sent through the Cryptographic API. For example, if 30 Miller-Rabin rounds are required, then issuing two 15 round Miller-Rabin requests would be an optimal usage of the Cryptographic API.

4.4.3.5 Large Number

Lookaside provides an interface to perform modular exponentiation and modular inversion functions. These are grouped together under the “Large Number” Category. These can be used as primitives for other cryptographic protocols. Large number operations are supported for all sizes up to a maximum of 4096 bits.

- Modular Exponentiation

Modular exponentiation involves taking an integer (the base), raising it to the power of another integer (the exponent) and then calculating the remainder left when this number is divided by the modulus. We calculate $result = base^{exponent} \bmod modulus$. The RSA and Diffie-Hellman operations both use specialized modular exponentiation which are optimized for those particular cases. For all other cases the “Large Number” implementation should be used.

- Modular Inversion

Modular inversion involves taking an integer (typically referred to as pA), inverting it (i.e. calculating 1/pA), and then calculating the remainder left when this number is divided by the modulus (typically referred to as pB). We calculate *result = (1/pA)* mod pB. This mod inv operation is generic and can be used by any application.

4.4.4 Lookaside Random Overview

The EP80579 integrated processor provides a Deterministic Random Bit Generator (DRBG) capability. Random numbers are used in many aspects of cryptography (for example as an initial IV for a cipher in CBC mode) and in the generation of prime numbers. Random number generation in combination with Primality testing can be used to create key material.

This feature can generate random bits that conform with the ANSI X9.82 part 1 specification.

§ §

5.0 QAT Access Layer Architecture Overview

5.1 What's New in this Chapter

No updates in this release.

5.2 Overview

The QAT Access Layer (QAT-AL) is responsible for management and configuration of the SSU and the driver firmware for the SSU running on the ASU. The QAT-AL component is initiated and started by the Acceleration System Driver (ASD) and stopped and shutdown afterwards also by the ASD.

After initialization of QAT-AL is executed, startup must be executed, followed by stop and then shutdown before QAT-AL can be initialized again.

The QAT-AL is responsible for:

1. Setup and test the entropy sample for Random Number Generation.
2. Setup communications structures for communication to and from the ASU (Acceleration Service Unit).
3. Sending the command messages to start and stop the firmware driver for the SSU (Security Services Unit).
4. Provide Version information and liveness of the SSU and the firmware driver of the SSU to the Debug Component.
5. Provide various statistics about the running of the SSU and communication rings.

The QAT-AL provides to other users:

1. A communications interface to communicate with the ASU.
2. A communication interface to allow other components to retest the entropy sample.

§ §

6.0 Debug Component Architecture Overview

6.1 What's New in this Chapter

- No updates in this release.

6.2 Overview

Debugging an application when problems occur can be difficult, especially when integrating with third-party software. To ease this burden, all of the EP80579 security software which runs in the Linux kernel is provided as source code, thereby facilitating debug as part of an application. In addition, the software provides mechanisms to support debug of the acceleration access layers and firmware modules. These mechanisms will help the end user in identifying and diagnosing the fault, whether they are due to defects in the software itself, or in the usage of this by the application.

As part of this debug infrastructure, the following debug features are supported:

- Version Information
- Liveness Detection
- Data Structure Dump (not supported in the current software release)
- Software Error Notification (SEN)

Each of these features is described in more detail below.

6.3 Version Information

To debug an issue, it is important to know the version of the software which is running. This version information consists of the following:

- Package version information that applies to a specific complete software package
- Individual software component version information for all the software components that are contained in the software package

The version information will contain the package/component name, the major number, minor number and patch number of the release.

6.4 Liveness Detection

Liveness detection is a mechanism to allow client applications to determine the runtime health of the various "threads of execution" that run within the Intel® EP80579 Integrated Processor. These may be threads running within the kernel on the IA core, or on the ASU. The client can at anytime query the "liveness" of all such threads of execution. The software will list all the threads of execution in the system, along with their state (dead or alive).

6.5 Data Structure Dump

Note: The Data Dump feature is not supported in the current software release.

Many of the software components in the Intel® EP80579 Integrated Processor maintain a certain amount of state information — debug counters, state variables, and so on — to help understand the current state of that software, and thereby debug or troubleshoot an application. This data is in addition to statistics maintained as part of the application (for example, counters required to support Management Information Bases, or MIBs). Data dump is a mechanism to retrieve these debug data structures.

Data dump provides a simple API with which a customer application can request the internal data structures and other information associated with the internal software. On receiving such a request for data dump information, the acceleration software gathers and provides necessary data to understand the internal state of the software.

On discovering an issue, the end-user should use this facility to dump the log and forward it to the Intel TME for further analysis of the problem.

§ §

7.0 ASD Module Architecture Overview

7.1 What's New in this Chapter

- No updates in this release.

7.2 Overview

The Acceleration System Driver is the kernel module responsible for initializing the Security subsystem on EP80579 integrated processor. It performs the following primary tasks:

- PCI driver for the Ring Controller and ASU cluster devices.
- loads firmware to the Acceleration Engines in the Acceleration Services Unit.
- provides hardware-related services for sub-component modules, for example, interrrupt management services.
- provides an interface to extract information set up by pre-boot firmware about non-coherent and coherent DRAM regions.
- controls the initialization and shutdown of the sub-component modules that make up the Security Subsystem.
- enables system resource variables to be modified using a user-space component that reads a configuration file at startup.

7.3 Functional Description

7.3.1 Configuration

Table 6 and Table 7 list the system resource variables.

Boot Time Configuration Instructions on page 34 provides more information.

Table 6. Cryptographic System Resource Variables

Parameter Name	Description	Default Value
NUM_CONCURRENT_LAC_SYMMETRIC_REQUESTS	Number of concurrent Cryptographic (LAC) symmetric requests allowed. Resources will be allocated during system initialization to support the specified number of concurrent requests. Specifically, this affects LAC Cipher, Hash and Combined Cipher and Hash requests, LAC Key and Mask Generation requests, and LAC Random Number Generation requests.	768
NUM_CONCURRENT_LAC_ASYMMETRIC_REQUESTS	Number of concurrent Cryptographic (LAC) asymmetric requests allowed. Resources will be allocated during system initialization to support this number. Specifically, this affects LAC Public Key Encryption requests for RSA, DSA, Diffie-Hellman, Prime Number Testing and Large Number Operations.	512
LAC_RANDOM_CACHE_SIZE	Size of a buffer used to preallocate LAC random numbers in a cache to use in synchronous mode.	131070

Table 7. Resource Variables

Parameter Name	Description	Default Value
ET_RING_LOOKASIDE_INTERRUPT_COALESCING_ENABLE	Enable interrupt coalescing on the ring.	1
ET_RING_LOOKASIDE_COALESCE_TIMER_NS	Frequency of coalesced interrupt for the ring in nanoseconds ***Note:*** The resolution of this timer is SKU-dependent with a minimum accuracy of 5 nanoseconds.	10000
ET_RING_MSI_INTERRUPT_ENABLE	Enable MSI interrupts for the ring controller. This setting improves performance, taking advantage of fully implemented APIC. 0 = Use INTx interrupts 1 = Use MSI interrupts (Default)	1
ET_RING_FAST_INTERRUPT_ENABLE	**Supported on FreeBSD only:** Enable FAST (filtered) Interrupts for the Ring Controller. This setting improves performance. Enable it only if the registered callback will not execute any potentially blocking operation. 0 = FAST interrupts disabled 1 = FAST interrupts enabled (Default)	1

7.4 Boot Time Configuration Instructions

A user space configuration program (asd_ctl) is included in the release package and is run automatically as part of the load script.

The configuration file /etc/icp_asd.conf is a simple configuration file containing the user-specified system resource variables and the values to be read into / set by the kernel in the configuration table.

The syntax for updating the configuration file is as follows:

```
# comment
token = value
token = value # comment
```

Blank lines are ignored and white spaces before and after a token or value are ignored.

Comments are denoted by the '#', any text read after a # symbol is ignored.

Note: If the same system resource variable is specified more than once in the configuration file, the last value read will be the one which is set in the configuration table.

A warning message will be printed if invalid syntax in the configuration file is encountered. The default values will be applied to all system resource variables that are not specified in the configuration file.

If an invalid value is specified for a system resource variable, a warning message will be printed and the default value will be applied.

Refer to Section 7.3.1 for a list of system resource variables that can be configured in the /etc/icp_asd.conf configuration file.

Example 1. Sample Configuration File

```
#icp_asd.conf example

#

NUM_CONCURRENT_LAC_SYMMETRIC_REQUESTS= 768

NUM_CONCURRENT_LAC_ASYMMETRIC_REQUESTS= 512

LAC_RANDOM_CACHE_SIZE= 131070

ET_RING_LOOKASIDE_INTERRUPT_COALESCING_ENABLE = 1

ET_RING_LOOKASIDE_COALESCE_TIMER_NS = 10000

ET_RING_MSI_INTERRUPT_ENABLE = 0
```

§ §

8.0 ASD Hardware Services

8.1 What's New in this Chapter

- No updates in this release.

8.2 Overview

This section describes the hardware-related services that the Acceleration System Driver (ASD) kernel module supplies to other modules.

The Acceleration System Driver controls the initialization and shutdown of other components that make up the Security Software Subsystem. Those components are:

- Hardware Access Layer (HAL)
- Debug Common Component (DCC)
- QAT Access Layer (QAT-AL)
- Lookaside Crypto Layer (LAC)

The initialization of the sub-components consists of a two stage process. Each sub-component provides an "init" function and a "start" function. The *init* function allocates resources required by the sub-component. The *start* function is provided that completes the initialization and finally enables the sub-component. ASD first invokes all sub-component *init* functions and then all the *start* functions. The ASD invokes them in the following order: HAL, DCC, QAT-AL, LAC.

The shutdown of the sub-components is the converse of the initialization mechanism and follows a similar two stage process. Each sub-component provides a "stop" function and a "shutdown" function. The *stop* function disables the sub-component. The *shutdown* function deallocates resources used by the sub-component.

8.3 Functional Description

This section describes the hardware-related services provided by ASD:

- Interrupt Management Services
- NCDRAM/CDRAM Interface

8.3.1 Interrupt Management Services

ASD registers an interrupt service routine with the Host OS, which enables:

- QAT-AL interrupt handler to process interrupts raised on rings 0-31

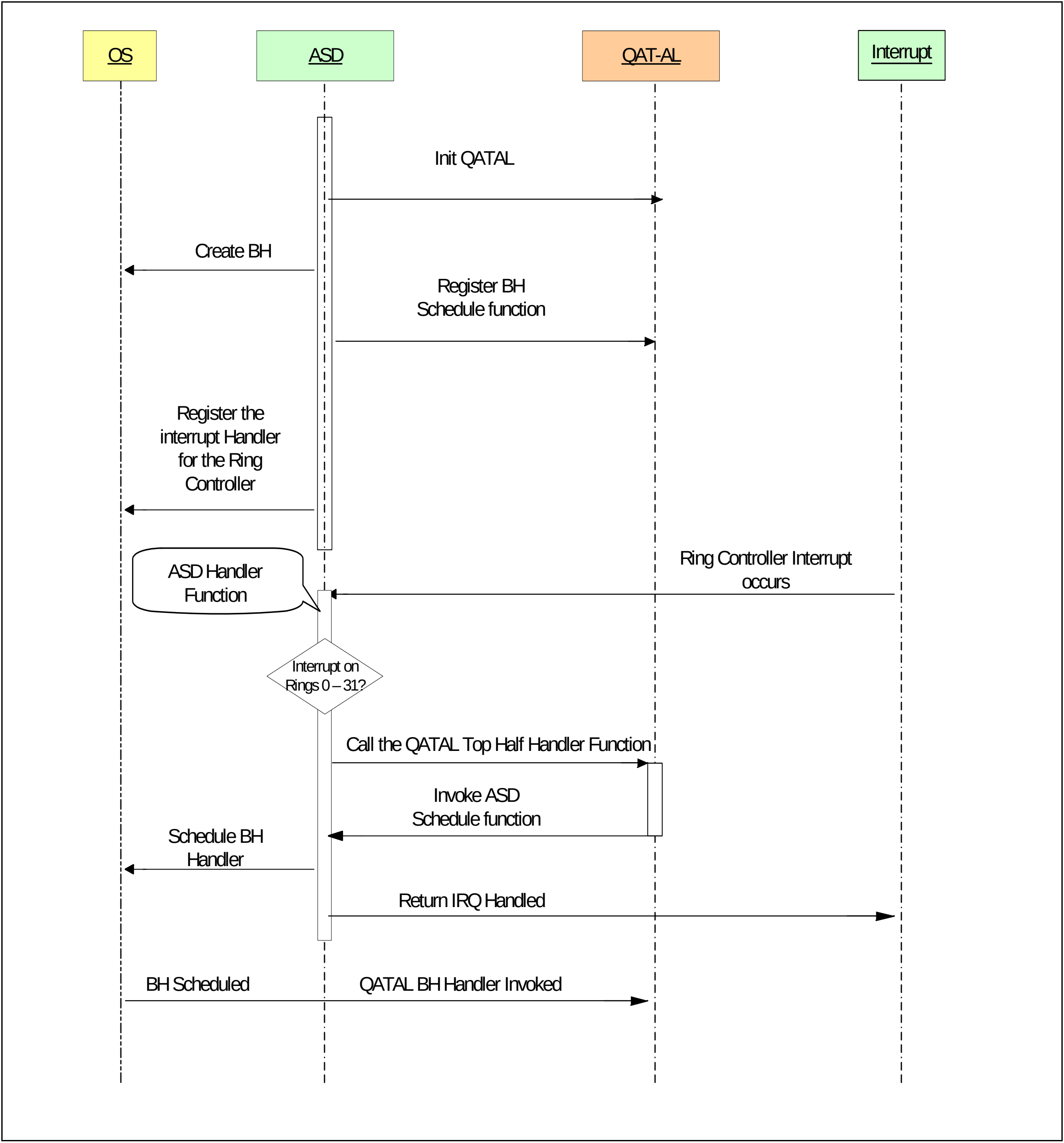
To enable ASD to provide these capabilities, the primitives listed in Table 8 are used:

Table 8. QAT-AL ISR Primitives

Function / Symbol	Description	Usage
`QatComms_intr`	This is the QAT-AL main ISR function	Used by ASD to bind to the Ring Controller IRQ for processing of interrupts on rings 0-31.
`QatComms_bh_handler`	This is the QAT-AL Bottom Half function	Used by ASD to bind to the Bottom Half Interrupt Handler for processing of interrupts on rings 0-31.
`QatComms_bh_schedule_register`	This function call enables the ASD register a function that will allow QAT-AL to schedule the Bottom Half Interrupt Handler	

The sequence diagram in Figure 7 illustrates how this operates:

Figure 7. ISR Sequence Diagram


OS
ASD
QAT-AL
Interrupt
Init QATAL
Create BH
Register BH Schedule function
Register the interrupt Handler for the Ring Controller
ASD Handler Function
Ring Controller Interrupt occurs
Interrupt on Rings 0 – 31?
Call the QATAL Top Half Handler Function
Invoke ASD Schedule function
Schedule BH Handler
Return IRQ Handled
BH Scheduled
QATAL BH Handler Invoked

8.3.2 NCDRAM/CDRAM Interface

8.3.2.1 Development Board Environment

The EP80579 with QuickAssist SKUs provide a direct non-coherent (NCDRAM) path between AIOC devices and the Memory Controller which is highlighted in red in Figure 8 on page 39.

The Software for Intel® EP80579 Integrated Processor product line uses three different memory regions, as defined in Table 9. The BIOS is responsible for setting up these regions. The IA/ASU Shared coherent and IA/ASU Shared AIOC-Direct (NCDRAM) regions allow the IA and ASU to manage a private pool of memory without OS involvement. For more information on these regions, refer to the Intel® EP80579 Integrated Processor Product Line Datasheet, Section 3.0.

The IA/ASU Shared coherent and NCDRAM requirements for a particular software package are defined in the [GET_STARTED_GD], "Coherent and Non-Coherent Memory Allocation" section.

Table 9. Memory Region Definitions

Datasheet Name	Software Name	Managed By	IA Cache Coherent††	Contents
IA O/S	IA O/S	O/S	Y	IA O/S and application code and data structures
IA/ASU Shared (Coherent)	CDRAM	EP80579 Driver†	Y	IA and AIOC shared data structures
IA/ASU Shared (AIOC-Direct)	NCDRAM	EP80579 Driver†	N	AIOC data structures; IA-32 core may access a portion via the EP80579 driver

† The EP80579 Driver includes the EP80579-specific software stacks that run on the IA, ASU, etc.
†† Indicates whether accesses to the region from the AIOC are coherent with the IA L2 cache.

The IA O/S, IA/ASU Shared coherent, and NCDRAM memory regions are allocated from the available system memory. The amount of DRAM available for allocation is located from address 0x00000000 to the value stored in the TopOfLowMemory (TOLM) register. The IA O/S, IA/ASU Shared coherent, and NCDRAM memory regions are allocated from available DRAM locations beginning with TOLM downward. NCDRAM memory space is allocated first, followed by IA/ASU Shared coherent, followed by IA O/S memory space. See Figure 9 on page 40 for more details.

Figure 8. Intel® EP80579 Integrated Processor with Intel® QuickAssist Technology Block Diagram

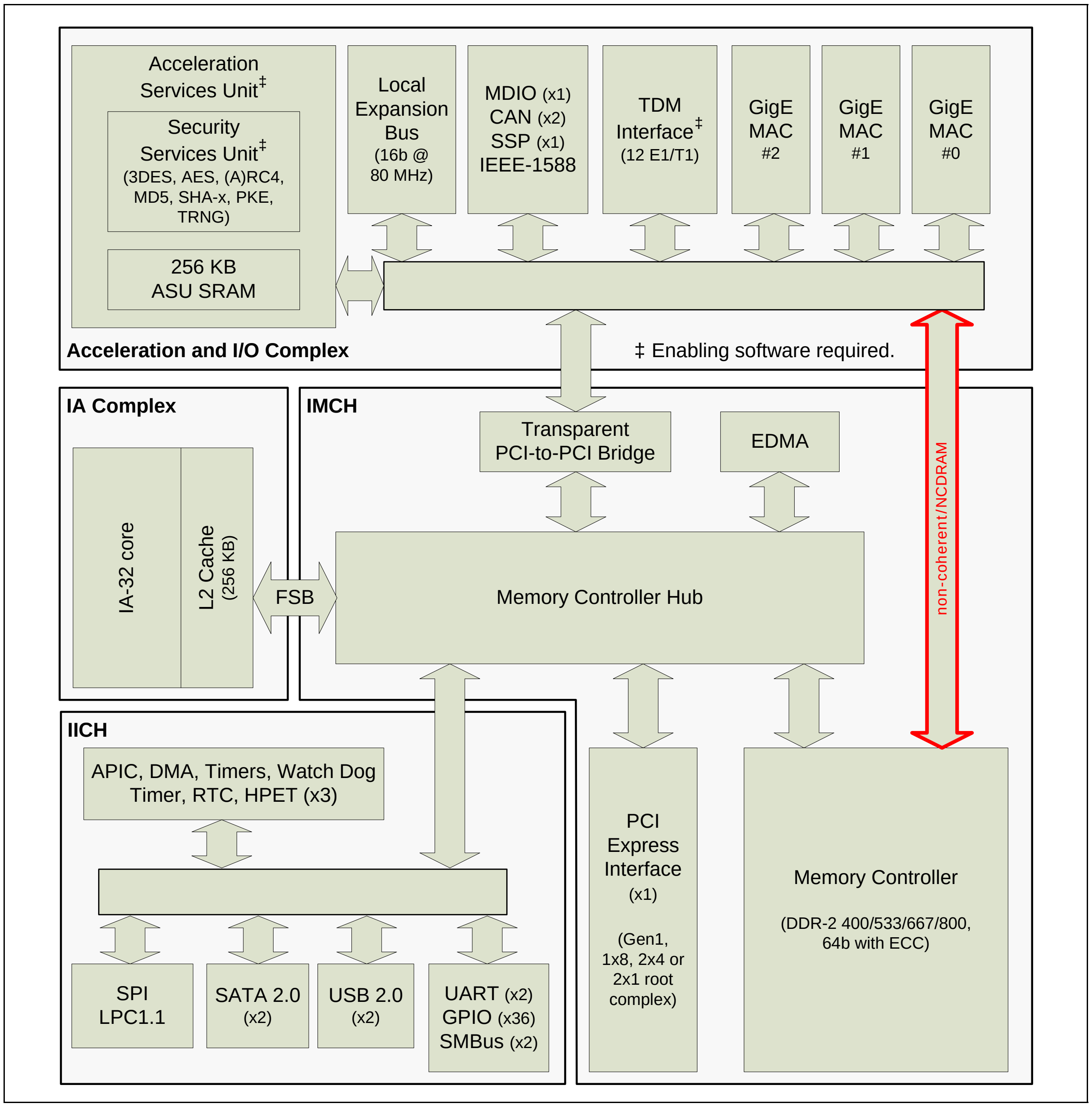
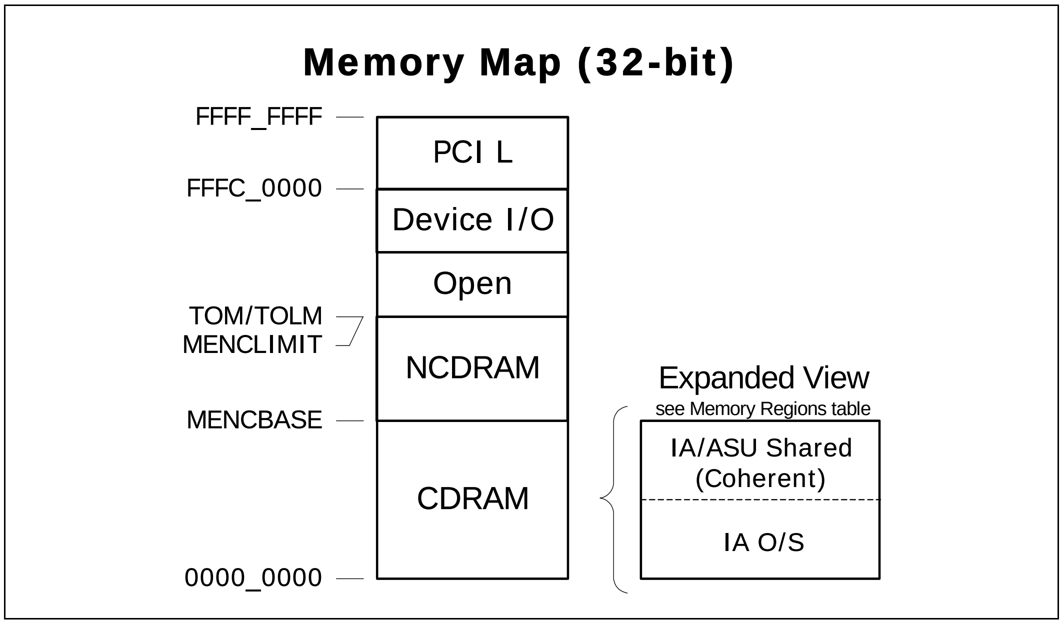

Acceleration Services Unit‡
Security Services Unit‡
(3DES, AES, (A)RC4, MD5, SHA-x, PKE, TRNG)
256 KB ASU SRAM
Local Expansion Bus (16b @ 80 MHz)
MDIO (x1)
CAN (x2)
SSP (x1)
IEEE-1588
TDM Interface‡ (12 E1/T1)
GigE MAC #2
GigE MAC #1
GigE MAC #0
Acceleration and I/O Complex
‡ Enabling software required.
IA Complex
IA-32 core
L2 Cache (256 KB)
FSB
IMCH
Transparent PCI-to-PCI Bridge
EDMA
Memory Controller Hub
non-coherent/NCDRAM
IICH
APIC, DMA, Timers, Watch Dog Timer, RTC, HPET (x3)
SPI LPC1.1
SATA 2.0 (x2)
USB 2.0 (x2)
UART (x2)
GPIO (x36)
SMBus (x2)
PCI Express Interface (x1)
(Gen1, 1x8, 2x4 or 2x1 root complex)
Memory Controller
(DDR-2 400/533/667/800, 64b with ECC)

Figure 9. Intel® EP80579 Integrated Processor Address Space



8.3.2.1.1 ACPI

ASD uses the ACPI mechanism to retrieve the memory region information set up by the pre-boot firmware.

This mechanism is dependent on an ACPI BIOS which supports four methods, each of which provides the equivalent to the EFI variables as outlined in Table 10.

Table 10. ACPI Shared RAM Methods

EFI Variable Name	ACPI Method
AccHWCoherentMemoryBase	method(PCMB, 0, serialized)
AccHWCoherentMemorySize	method(SCMB, 0, serialized)
AccHWnonCoherentMemoryBase	method(PNMB, 0, serialized)
AccHWnonCoherentMemorySize	method(SNMB, 0, serialized)

The ASD device driver evaluates the ACPI methods to retrieve the pointer and sizes of the memory allocated in the BIOS. Each OS supports different interfaces to evaluate the ACPI methods.

§ §

Part 2: Using the API

This part of the document provides an overview of how to use the EP80579 security software acceleration APIs to build an application.

Individual APIs are described in their corresponding reference manuals, which are listed in Table 1, "Related Documents and Sample Code" on page 8. The reference manuals contain details of the data structures, data types, function signatures, and other detailed constructs to allow you to call individual functions correctly.

This part of the document is where you look to understand how to string such calls together to do something useful. The chapters in this part of the document are organized as follows:

- Chapter 9.0, "Introduction to Use Cases" gives an introduction to building an application using the Acceleration APIs.
- Chapter 10.0, "Programming Model" describes the programming model which is common to all of the Acceleration APIs. This includes an overview of the coding conventions, as well as the invocation model for the asynchronous APIs.
- Chapter 11.0, "Debugging Applications"
- Chapter 12.0, "Using the Intel® QuickAssist Technology Cryptographic API"
- Appendix A, "NPF Copyright Notice"

§ §

9.0 Introduction to Use Cases

This chapter discusses what's involved in building an application using the Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology.

9.1 What's New in this Chapter

Added Note in Section 9.2.1 explaining cryptographic framework "shim" support.

9.2 Use Cases

There are a wide variety of security applications that can be developed using the Intel® EP80579 Software for Security Applications on Intel® QuickAssist Technology.

Each of these use cases will be described in more detail in later chapters within this document.

9.2.1 Lookaside Acceleration Model

Key use cases for the lookaside acceleration model are:

- A client may wish to use the Cryptographic API to accelerate cryptographic operations. This may be with a view to accelerating the encryption/decryption of "data at rest" — for example, files in a file system — or for protecting "data on the move" — for example, packets in a network. An existing application can be modified, or new code can be written from scratch, to use the Cryptographic API.
- In other cases, a client may be using a cryptographic framework, such as the OpenBSD* Cryptographic Framework (OCF). Intel supplies a driver which enables the Cryptographic features to be accessed via OCF; this driver is sometimes referred to as the "OCF shim", or adapter. In this way, the cryptographic operations are transparently and seamlessly accelerated.

  Note: The EP80579 security software release package version 1.0.3 does not support OpenBSD/FreeBSD Cryptographic Framework (OCF), OCF-Linux, or any open source projects such as Openswan*, OpenSSL*, or Racoon*. If your application requires OCF, you must use security software package version 1.0.2 which includes shim software to enable OCF support.
- A client may wish to accelerate a cryptographic protocol, such as IPSec. The open-source Openswan* project is an implementation of IPSec that can use OCF for implementing cryptographic operations. By using the "OCF shim" described above, a client can utilize the seamless acceleration of the cryptographic operations.

See Chapter 12.0, "Using the Intel® QuickAssist Technology Cryptographic API" for a more detailed description of these usage models.

§ §

10.0 Programming Model

10.1 What's New in this Chapter

No updates in this release.

10.2 Overview

There are two different "categories" of API supplied with EP80579 integrated processor software, as follows:

- APIs which are part of the Intel® QuickAssist Technology program. The only API in this software release which falls into this category is the [CRYPTO_API]. The set of conventions governing these APIs are documented in Section 10.3, "Intel® QuickAssist Technology API Conventions" on page 43.
- "Other" APIs, which are **not** part of the Intel® QuickAssist Technology program. The only API in this software release which falls into this category is the [DEBUG_API]. The set of conventions governing these APIs are documented in Section 10.4, "Other API Conventions" on page 45.

10.3 Intel® QuickAssist Technology API Conventions

Note: This section discusses conventions for APIs which are part of the Intel® QuickAssist Technology program, such as [CRYPTO_API].

10.3.1 Memory Allocation and Ownership

The convention is that all memory needed by an API implementation is allocated outside of that implementation. In other words, the APIs are defined such that the memory needed to execute operations are supplied by a client entity or platform control entity rather than having memory allocated internally.

Memory used for parameters are owned by the side (caller or callee) that allocated them. An owner is responsible for de-allocating the memory when it is no longer needed. If an API has an allocation function, it shall also have a symmetric de-allocation function. The caller of the allocation function acts as the owner and is responsible for invoking the appropriate de-allocation routine when the memory is no longer needed.

Generally, memory ownership does not change. For example, if a program allocates memory and then passes a pointer to the memory as a parameter to a function call, the caller retains ownership and is still responsible for de-allocation of the memory. This is the default behavior and any function which deviates from this behavior must clearly state so in the function definition.

10.3.2 Data Buffer Models

Data buffers that are passed across the API interface in one of the following formats:

- Flat Buffers represent a single region of physically contiguous memory, and are described in Section 10.3.2.1, "Flat Buffers" on page 44.
- Scatter Gather Lists are essentially an array of flat buffers, for cases where the memory is not all physically contiguous. These are described in Section 10.3.2.2, "Scatter Gather Lists" on page 44.

10.3.2.1 Flat Buffers

Flat buffers are represented by the type CpaFlatBuffer, defined in the file cpa.h. It consists of two fields:

- data pointer which points to the start address of the data or payload. This is a virtual address. The data to which this points is required to be in contiguous physical memory.
- length of this buffer

For additional details, see Section 12.4.1, "Flat Buffers" on page 64.

10.3.2.2 Scatter Gather Lists

A scatter gather list is defined by the type CpaBufferList, also defined in the file cpa.h. The buffer list contains four fields, as follows:

- number of buffers in the list
- pointer to an unbounded array of flat buffers
- user data: an opaque field and is not read or modified internally by the API
- pointer to meta data required by the API: The meta data is required for internal use by the API. The memory for this buffer needs to be allocated by the client as contiguous data. The size of this meta data buffer is obtained by calling the appropriate *GetMetaSize* function.

For additional details, see Section 12.4.2, "Buffer List" on page 65.

10.3.3 Synchronous and Asynchronous Support

The Cryptographic API may be called in either asynchronous or synchronous modes.

10.3.3.1 Asynchronous Operation

The caller specifies asynchronous mode by supplying a callback function to the API. Control returns to the client once the request message has been sent to the SSU; the function does not block. The callback is invoked in a bottom half context (on Linux, this is a tasklet) when the SSU completes the operation.

This mode is preferred for optimal performance.

10.3.3.2 Synchronous Operation

The caller specifies synchronous mode by not supplying a callback function pointer; a NULL function pointer is passed. Once the request message has been sent to the SSU, the thread of execution blocks pending receipt of the response, or a timeout. Once a response is received from the SSU, it is processed, and the calling thread is unblocked and resumes processing.

Because it is blocking, synchronous mode should not be used in contexts where blocking is not allowed, for example in interrupt context on Linux.

10.3.4 Pre-Registration

In a number of accelerator use cases, the concept of a session applies. A session consists of a setup, multiple data phases and a tear down. The data phases are typically long lived (with respect to session setup time); as a result there is a class of API that requires registration. For example, symmetric cryptographic operations can benefit from pre-registration. Data which is common to all operations can be provided and/or pre-computed at session initialization time.

Alternately, there are scenarios where all the required data for a call are available at the time of the call. These do not require pre-registration of a session.

10.4 Other API Conventions

Note: This section discusses conventions for APIs which are **not** part of the Intel® QuickAssist Technology program, such as [DEBUG_API].

The “other” Acceleration APIs use some common conventions and share a common programming model, which is loosely based on [NPF API]. Some of the key differences between the NPF conventions and those adopted by the Acceleration APIs are summarized below.

- Callback registration is not required, as described in Section 10.4.1
- Memory allocation and ownership is clarified, as described in Section 10.4.2
- Callback data structures are clarified, as described in Section 10.4.3

10.4.1 Asynchronous API and Function Completion Callbacks

Similar to what is described in Section 7 of [NPF API], most of the Acceleration APIs are asynchronous. Completion of the work associated with an API function call is indicated not by the return of the function, but by the invocation of a separate completion callback function by the callee to the caller. This can be thought of as a request/response mechanism. Such an asynchronous API allows for greater parallelism to be achieved, while still allowing synchronous behavior to be easily layered on top of the asynchronous callbacks if desired.

This asynchronous request/response model is suitable for an architecture such as the EP80579 security software’s, where requests to perform cryptographic or other operations can be sent to dedicated on-chip accelerators, and then responses received when the operation is complete. The requests and responses are typically sent via rings, as described in Chapter 2.0, “Silicon Overview.”

The key differences between the conventions adopted by the Acceleration APIs versus those described in [NPF API] are as follows:

- Callback registration (and de-registration) is not required. The complexity imposed on applications to manage callback handles was judged to outweigh any potential benefits. As a result of this:
    - — Instead of passing a callback handle to every asynchronous function on the Acceleration APIs, a function pointer is passed in.
    - — There is only one type of application context information, namely the correlator which is provided at API function call time. The user context, which is provided at callback registration time in the NPF model, is not required.

- There is a 1:1 relationship between an asynchronous request call and a completion callback (response). [NPF API] allows for 1:N relationships here. The complexity associated with managing multiple responses per callback was judged to outweigh the potential performance benfit of coalescing responses. As a result of this:
  - — There is only one asynchronous response per callback.
  - — There is no need for an overall status.
  - — There is no need for the callback data to contain a number/array of asynchronous responses.

  For a summary of what is contained within this callback data structure, see Section 10.4.3, “Callback Data Structures” on page 46.

Note that the context in which callbacks are invoked is described in Section 3.5, “Process View” on page 18.

10.4.2 Memory Allocation and Ownership

[NPF API], Section 6.4, briefly discusses memory ownership. To make the issue clearer, the following conventions have been adopted.

For all output parameters on asynchronous API functions, the following memory allocation model applies:

- Memory should be allocated by the client
- The memory is passed into the Acceleration API function as a pointer.
- The pointer is returned in the callback data structure, which is a parameter to the completion callback function.
- Memory should be freed by the client.

This means that ownership is temporarily granted to the asynchronous API implementation, and reverts to the client only after the function completion callback function is invoked. See Section 10.4.1, “Asynchronous API and Function Completion Callbacks” on page 45 for details.

In some of the EP80579 security software APIs, there are numerous input and output parameters, and/or multiple parameters within a single data structure, some of which are inputs and some of which are outputs. To make the memory ownership clear in these cases, all memory for which ownership is temporarily granted to the API implementation should be grouped into a single data structure. By convention, this data structure has the suffix “_op_data_t” (for operation data).

Where there is a single output parameter, this struct is typically not created.

Any exceptions to this model are documented by the corresponding APIs. For example, in some cases, the ownership of a data structure is retained by the API implementation even after the function completion callback has been invoked.

10.4.3 Callback Data Structures

The function completion callback function takes the following parameters:

- Correlator (discussed in Section 10.4.1, “Asynchronous API and Function Completion Callbacks” on page 45)
- Callback data structure, which contains the following information:
  - — Status of the requested operation

— Operation type. For the typical case, where the same callback data structure type is used for multiple different functions (operation types) on a given API, this is used to distinguish the function for which this is the response.

— Operation-specific data. This may be the data structure described in Section 10.4.2, "Memory Allocation and Ownership" on page 46 with the suffix _op_data_t, or it may be a single output parameter. In the case where there are multiple different types of operation-specific data, depending on the operation type, then all are provided within a union construct; the operation type can be used to decide which field of the union is valid.

10.4.4 Return Codes

Note: This section discusses error values for APIs which are **not** part of the Intel® QuickAssist Technology program, such as [DEBUG_API].

In the case of an error occurring when an API function is invoked, the API function will return with the error condition and no function completion callback function will be invoked.

Table 11 shows the error values that are defined in icp.h, along with their descriptions.

Table 11. Error Values for Other APIs

Value	Description
ICP_E_NO_ERROR	Success status.
ICP_E_FAIL	Fail status.
ICP_E_RETRY	Retry status. Indicates a temporary condition exists which prevents the request from being completed at this time, usually due to the limited capacity of some internal resource being exhausted such as internal message queue is full, internal buffer pool is empty, etc. Recommended approach is to wait for a short amount of time (1 millisecond or less) and submit the request again. This is expected behavior, but may be an indication that some system-level tuning is required, for example, to increase internal resource limits or to throttle the rate of new request submissions.
ICP_E_UNDERFLOW	Underflow error - client is under submitting data.
ICP_E_OVERFLOW	Overflow error - client is over submitting data.
ICP_E_INVALID_PARAM	Invalid parameter passed in.
ICP_E_NULL_PARAM	One or more parameters is null.
ICP_E_MUTEX	Failure with a mutex operation.
ICP_E_RESOURCE	Error related to system resource.
ICP_E_FATAL	A serious error has occurred. Recommended course of action is to shutdown and restart the component.
ICP_E_ALREADY_REGISTERED	An attempt was made to register (for example a callback) with the same "key" value as an existing registration.
ICP_E_INVALID_HANDLE	An invalid handle was passed in.
ICP_E_NOT_SUPPORTED	Operation not supported in the current implementation.

§ §

11.0 Debugging Applications

11.1 What's New in this Chapter

- No updates in this release.

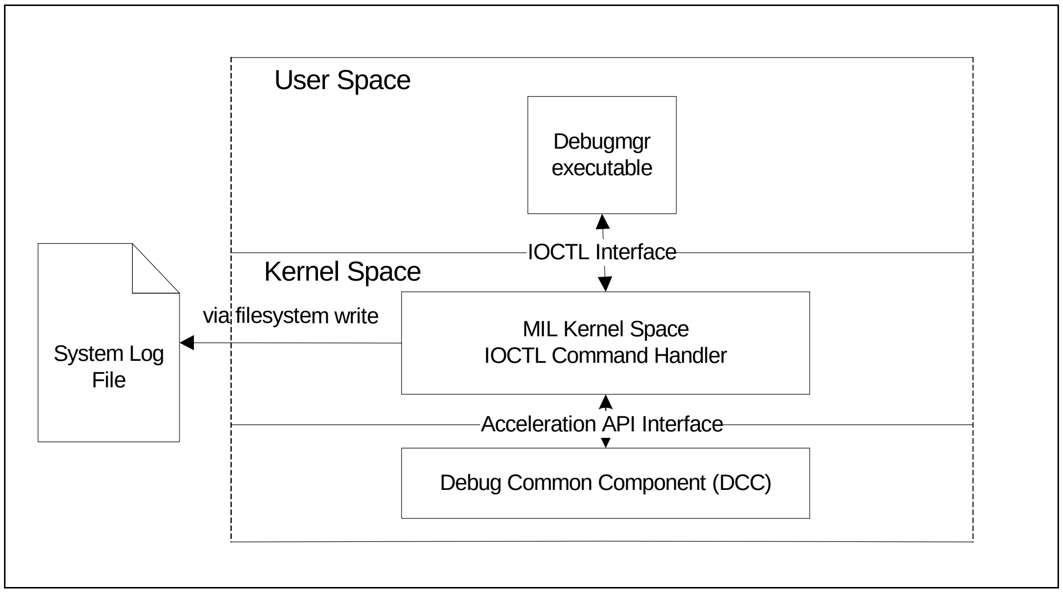
11.2 Management Interface Layer (MIL) Introduction

A reference application called Management Interface Layer (MIL) is provided in order for the users to make use of the DCC APIs provided. MIL provides a framework for triggering and logging of various debug features in a uniform and generic manner. The MIL application by itself does not generate any debug information. MIL makes use of the APIs exposed by the DCC component and generates a standard Linux* log file called syslog.

Note: It is the responsibility of the programmer to ensure the syslog maximum size is > 4 k messages or debug messages may be lost.

Figure 10 illustrates debug components and how they fit together:

Figure 10. Management Interface Layer Architecture Decomposition


User Space
Debugmgr executable
IOCTL Interface
Kernel Space
via filesystem write
System Log File
MIL Kernel Space IOCTL Command Handler
Acceleration API Interface
Debug Common Component (DCC)

The MIL application provides user space commands in order to enable/disable debug logging as well as trigger logging for Version, Liveness, Data dump as well as SEN events.

The MIL application is invoked from the command line using the following syntax:

`debugmgr {command}`

where `{command}` is one of the following:

- — help
- — DebugEnable
- — DebugDisable
- — VersionDumpAll
- — setHC <timeout>
- — SystemHealthCheck
- — DataDump (not supported in this release)
- — SetFileName <filename>

These commands are described in more detail in Section 11.3.

11.2.1 Loading the MIL Application

On FreeBSD, load the debug manager using the following commands:

```
cd /EP805XX_release
setenv ICP_ROOT $PWD
kldload $ICP_ROOT/StagingArea/icp_debugmgmt.ko
```

On Linux, follow these steps:

1. Load the debug manager using the following commands:

```
cd /EP805XX_release
export ICP_ROOT=$PWD
insmod $ICP_ROOT/StagingArea/icp_debugmgmt.ko
```

2. Create a character device /dev/mil_driver using the following steps:

   a. Find the device major number associated with the character device /dev/mil_driver using the following command:

   ```
   cat /proc/devices | grep mil_driver
   ```

   The following is a sample output from the above command:

   ```
   249 /dev/mil_driver
   ```

   b. Use the device major number displayed by the above command to create the character device using the following command:

   ```
   mknod /dev/mil_driver c <device major number> 0
   ```

   If we use the sample output from step a., then the mknod command would be:

   ```
   mknod /dev/mil_driver c 249 0
   ```

11.3 MIL User Command Details

Note: The debugmgr must be enabled using the option ‘DebugEnable’ before it can display information and must be disabled using the option ‘DebugDisable’ at the end of it.

11.3.1 help

The help command lists all the user space commands available.

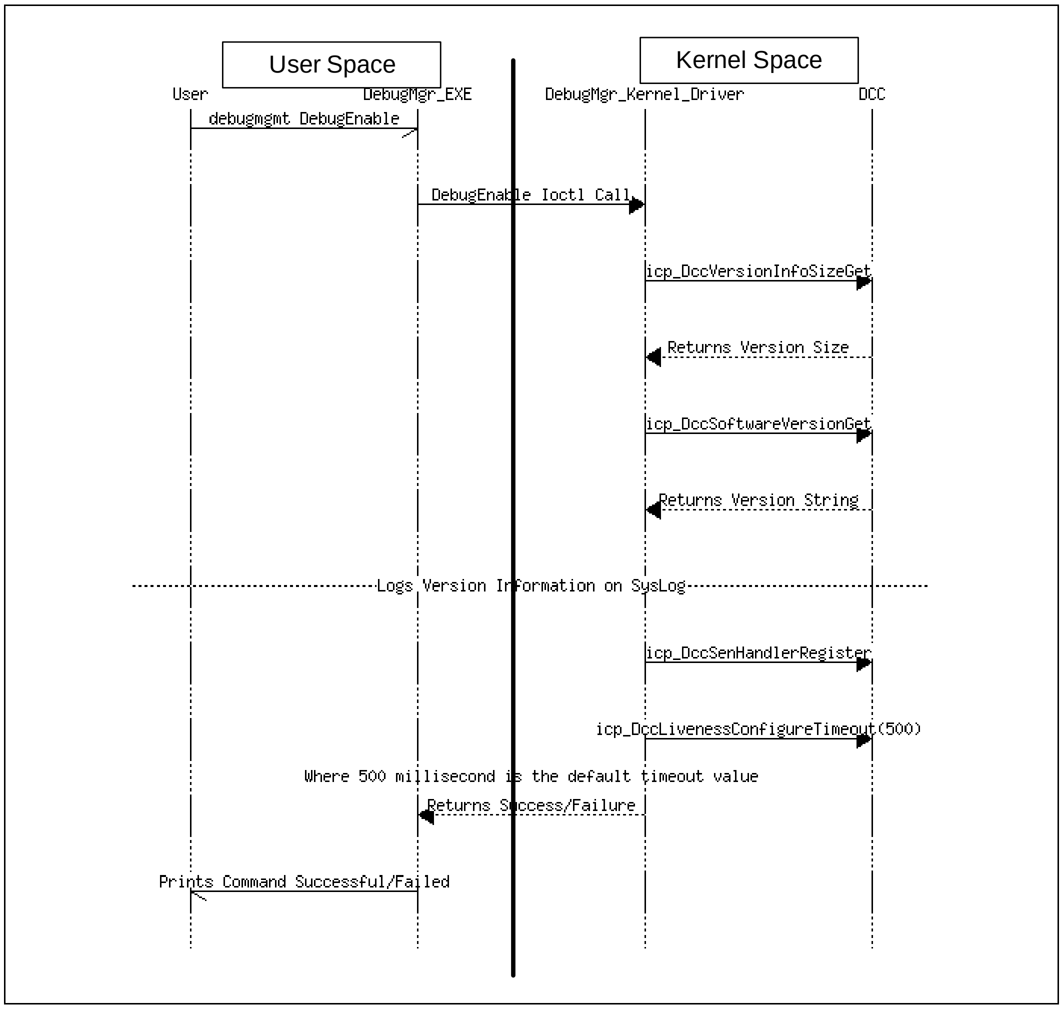
11.3.2 DebugEnable

The DebugEnable command enables the debug facility within the EP80579 security software acceleration subsystem. This command must be invoked before any other command is invoked. Specifically, DebugEnable does the following:

- reports the version information for all components within the software stack, and logs this information to the system log in a well-defined format.
- registers for all System Error Notifications. The SEN handler simply logs the errors to the system log.
- sets a default value (500 ms) for the "liveness" polling interval.

See Figure 11 for an illustration of this behavior.

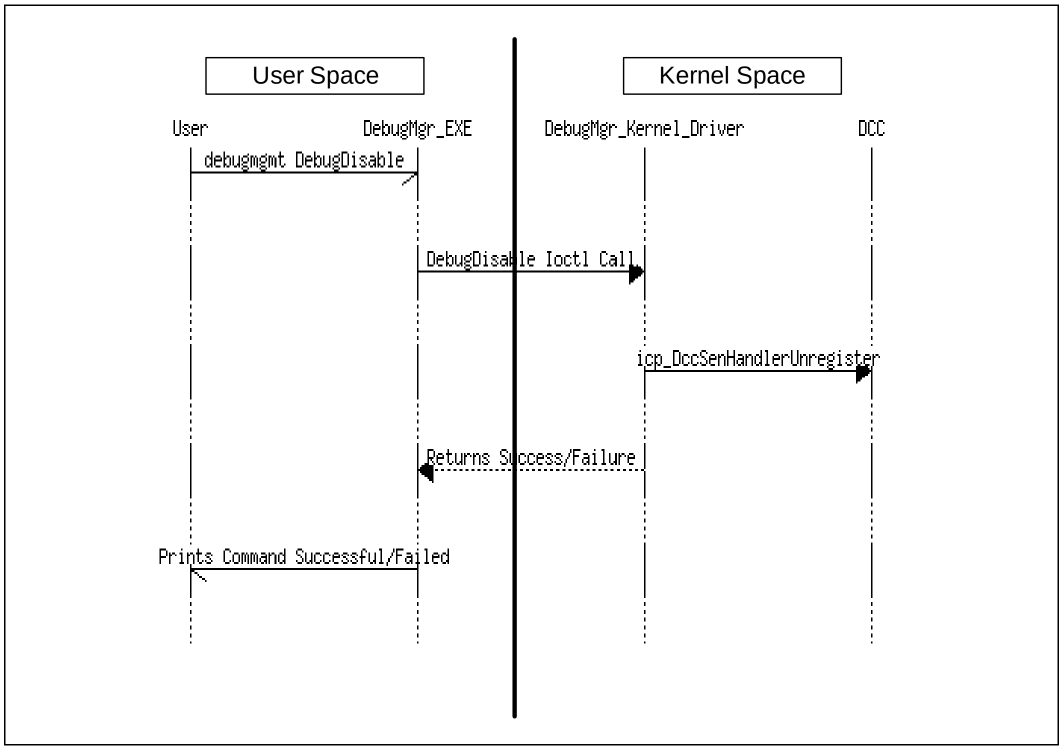
Figure 11. Sequence Diagram for DebugEnable Command


User Space
Kernel Space
User
DebugMgr_EXE
DebugMgr_Kernel_Driver
DCC
debugmgmt DebugEnable
DebugEnable Ioctl Call
icp_DccVersionInfoSizeGet
Returns Version Size
icp_DccSoftwareVersionGet
Returns Version String
Logs Version Information on SysLog
icp_DccSenHandlerRegister
icp_DccLivenessConfigureTimeout(500)
Where 500 millisecond is the default timeout value
Returns Success/Failure
Prints Command Successful/Failed

11.3.3 DebugDisable

The DebugDisable command disables the debug facility within the EP80579 security software acceleration subsystem. Specifically, it unregisters the SEN handlers registered at DebugEnable time. It then disables further command invocation other than DebugEnable from the user side.

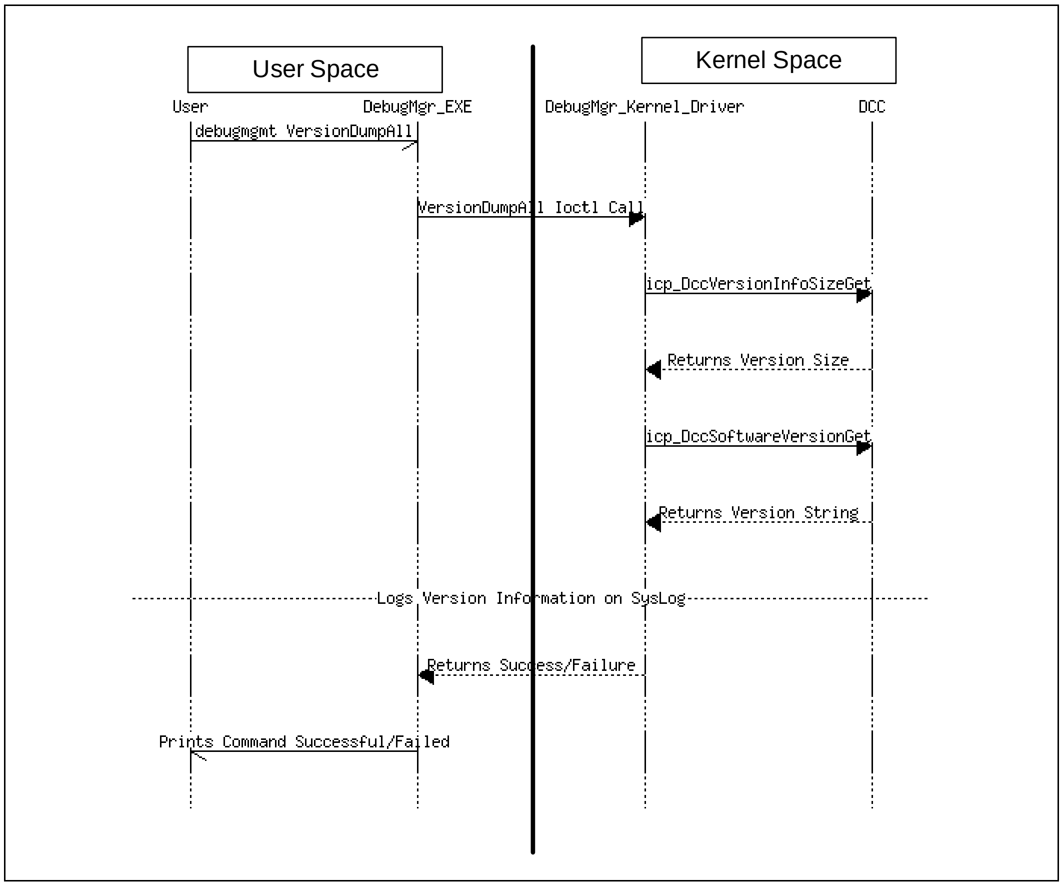
Figure 12. Sequence Diagram for DebugDisable Command



11.3.4 VersionDumpAll

This command results in the version of all components within the software stack being logged to the syslog file as specified in Figure 13.

Figure 13. Sequence Diagram for VersionDumpAll Command



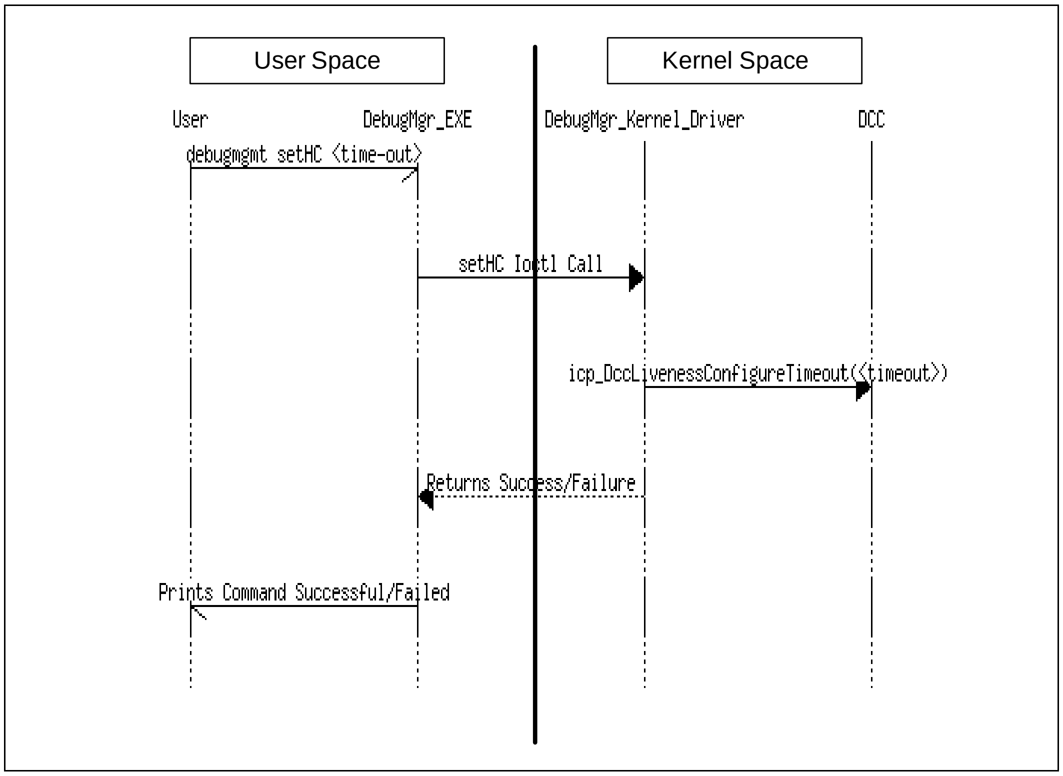
11.3.5 setHC <timeout>

This command is used to specify the maximum time interval for any one thread to punch the liveness value. When the timeout is exceeded, a thread that has not indicated it is alive will be reported as dead. After getting all the timeout values for each of the threads in the system, the system application needs to define this value and set it appropriately.

The valid range for <timeout> is from 100 to 5000 milliseconds.

Note: If the value is too small, it is possible that false reports of dead threads will be received.

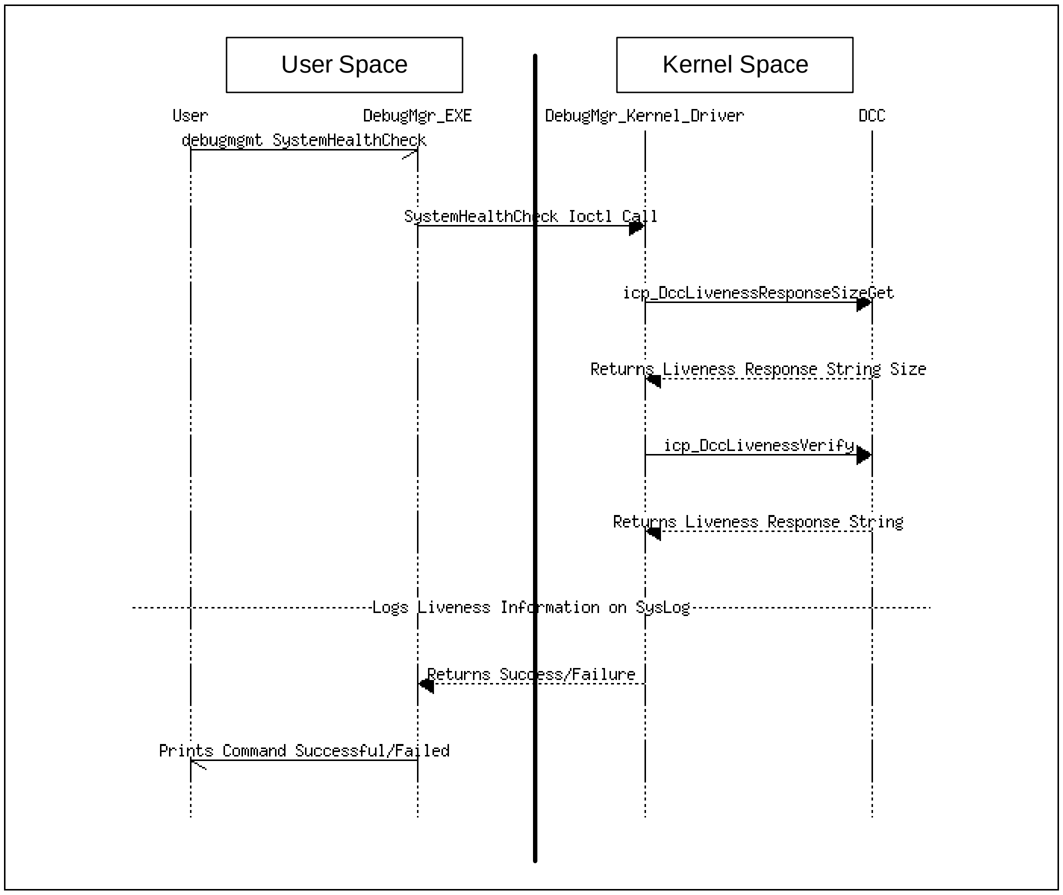
Figure 14. Sequence Diagram for setHC Command



11.3.6 SystemHealthCheck

This command causes the liveness of each active thread in the system to be queried and displays this information in the syslog file as shown in Figure 15.

Figure 15. Sequence Diagram for SystemHealthCheck Command

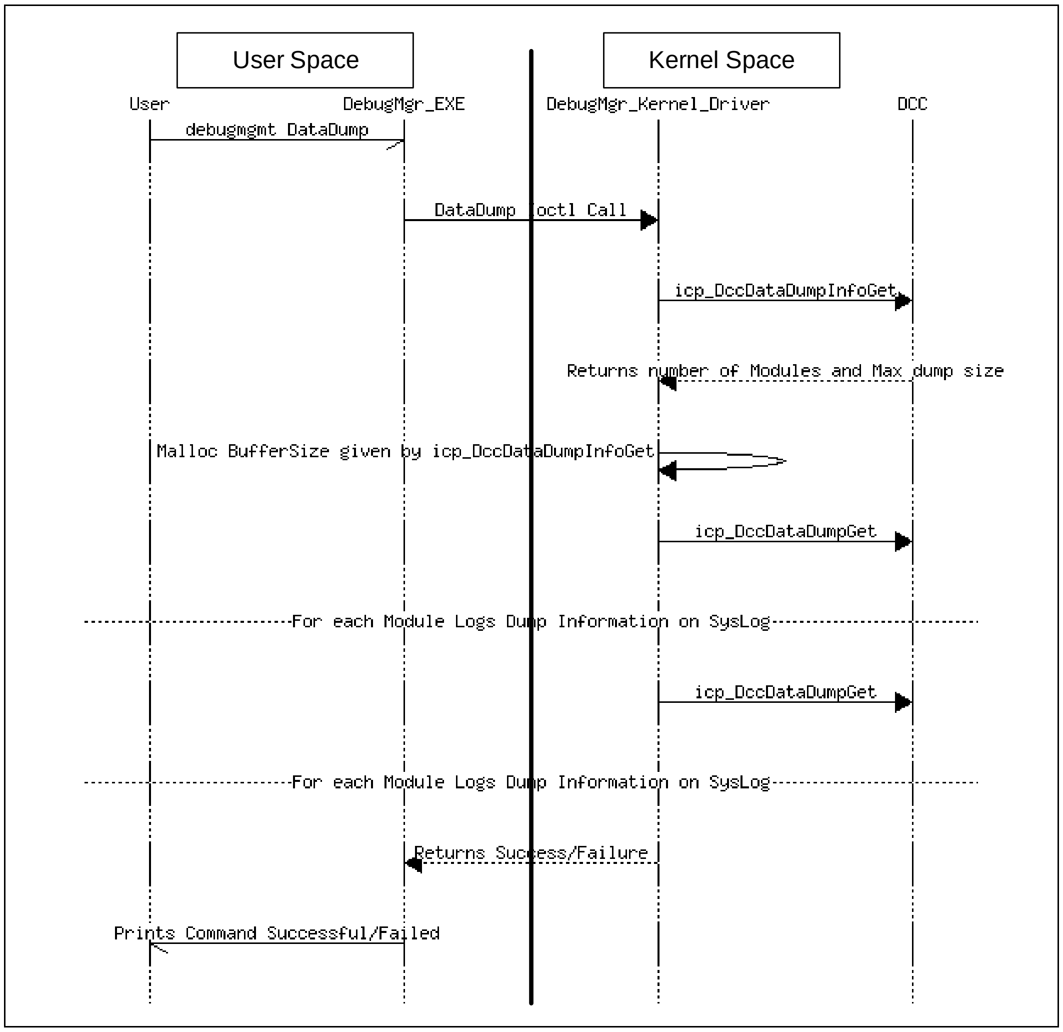

User Space
Kernel Space
User
DebugMgr_EXE
DebugMgr_Kernel_Driver
DCC
debugmgmt SystemHealthCheck
SystemHealthCheck Ioctl Call
icp_DccLivenessResponseSizeGet
Returns Liveness Response String Size
icp_DccLivenessVerify
Returns Liveness Response String
Logs Liveness Information on SysLog
Returns Success/Failure
Prints Command Successful/Failed

11.3.7 DataDump

Note: The Data Dump feature is not supported in the current software release.

This command is used to generate a data dump from the entire acceleration subsystem. The Debug API is queried to determine the maximum amount of memory required by any one component. This amount of memory is then allocated, and passed in turn to each component to dump its data, which is then written to the system log.

Figure 16. Sequence Diagram for DataDump Command

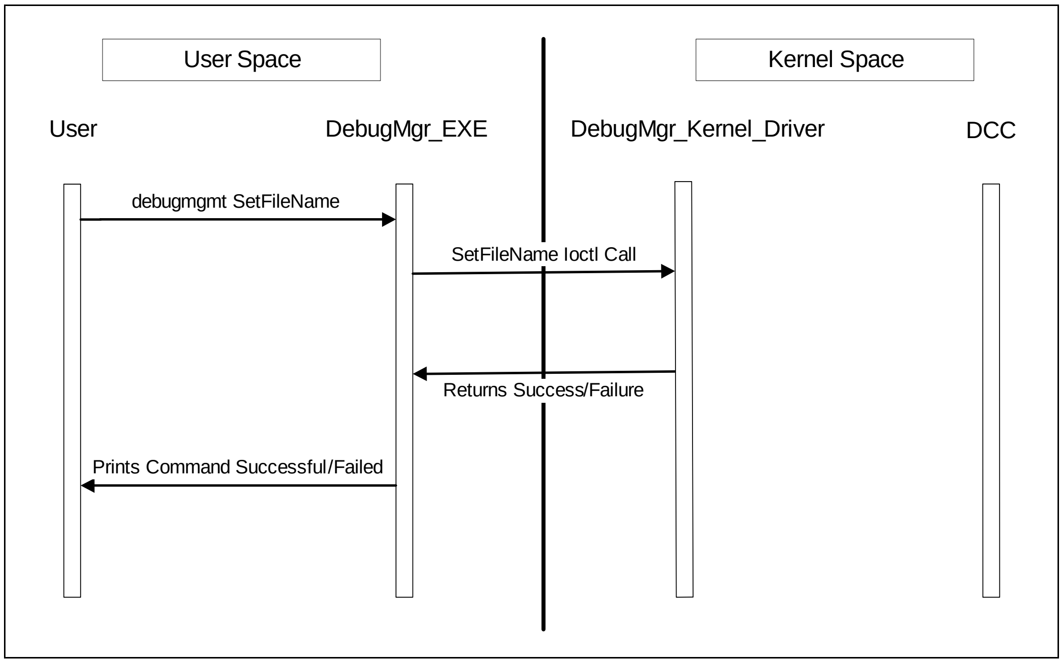

User Space
Kernel Space
User
DebugMgr_EXE
DebugMgr_Kernel_Driver
DCC
debugmgmt DataDump
DataDump ioctl Call
icp_DccDataDumpInfoGet
Returns number of Modules and Max dump size
Malloc BufferSize given by icp_DccDataDumpInfoGet
icp_DccDataDumpGet
For each Module Logs Dump Information on SysLog
icp_DccDataDumpGet
For each Module Logs Dump Information on SysLog
Returns Success/Failure
Prints Command Successful/Failed

11.3.8 SetFileName <filename>

This command is used to set the name of the system log file which contains all the debug information. The maximum length of the <filename> is 256 characters.

Note: If this command is not called, by default, the information will be stored in: /var/log/icp_debugmgmt.log

Figure 17. Sequence Diagram for SetFileName Command



11.4 APIs

Table 12. Debug APIs (Sheet 1 of 2)

Name	Description
icp_DccVersionInfoSizeGet	This function provides the DCC Client with buffer size needed to retrieve Version information of all the Package/Components registered with DCC. The user should allocate this buffer and pass it to DCC to retrieve the Package and Components version information.
icp_DccSoftwareVersionGet	This function provides the caller with version information for the Package and each of the Components registered with the DCC. This function should be called with the required amount of buffer allocated for the returned information.
icp_DccLivenessConfigureTimeout	This API is used to configure the timeout period (in ms) for response monitoring. The DCC waits for a response during this period before declaring a thread to be alive or dead. Note that the timeout period should be large enough to allow each of the threads to individually respond within this period. The timeout value should be defined to accommodate threads with large workloads.

Table 12. Debug APIs (Sheet 2 of 2)

Name	Description
icp_DccLivenessResponseSizeGet	The user calls this API to get the buffer size to be allocated for the retrieval of system response information. The user should free this buffer after the buffer information is processed.
icp_DccLivenessVerify	The user calls this API to verify response of all threads of execution in the system. The user provides the required buffer which the DCC fills with Response status information.
icp_DccDataDumpInfoGet	This function returns the number of Modules and the maximum size of buffer needed for any one dump query transaction. Before invoking the Data Dump query request, the user gets the maximum size of the buffer to allocate and the total number of Modules that should be queried for the data dump.
icp_DccDataDumpGet	The user should use this function to dump the data structures of Modules registered with the DCC. This function will be called once for each registered Module for the data dump information. With each call, the user passes to DCC a pre-allocated buffer which is passed to the Module to be filled with the dump information.
icp_DccSenHandlerRegister	The user registers the SEN event callback handler with this function. When a SEN event occurs, the DCC calls the handler registered in this prototype format. ***Note:*** If no callback is registered by the user, then it will default to syslog.
icp_DccSenHandlerUnregister	The user unregisters the SEN event callback handler.

§ §

12.0 Using the Intel® QuickAssist Technology Cryptographic API

The Intel® QuickAssist Technology Cryptographic API for the Lookaside Model is described in the API user reference document [CRYPTO_API]. Most of the API functions support both asynchronous and synchronous invocation, except for those used to initialize and remove sessions and for getting statistics which support synchronous invocation only. This chapter provides a brief overview of the API.

12.1 What's New in this Chapter

- Section 12.9 and Section 12.10: New Note to explain shim support

12.2 Intel® QuickAssist Technology Cryptographic API

The API is documented in the [CRYPTO_API] document. The API can be split into three broad areas. These are as follows:

- **Common**: The cpa.h, cpa_cy_common.h and icp_lac_cfg.h files define common API, enums and so on, which both the symmetric and asymmetric APIs use.

Table 13. Cryptographic Common Interface Summary (icp_lac_cfg.h)

Method	Description
icp_AsdCfgLacInit	Initialize the Cryptographic Service setting up all static data tables.
icp_AsdCfgLacShutdown	Shutdown the Cryptographic Service, therefore cleaning up all resources owned by the module
icp_AsdCfgLacStart	Start the Cryptographic services, it is assumed at this point Cryptographic has been initialized.
icp_AsdCfgLacStop	Stop the Cryptographic services.
Note: These APIs are called by the ASD component to ensure correct initialization/shutdown procedures.	

- **Instance Management**: The cpa_cy_im.h file defines the functions for managing instances.

Table 14. Cryptographic Instance Management Summary (cpa_cy_im.h)

Method	Description
cpaCyStartInstance	This function will initialize and start the Cryptographic component.
cpaCyStopInstance	This function will stop the Cryptographic component and free all system resources associated with it.

- **Symmetric**: The cpa_cy_sym.h file contains the symmetric API for hashing, cipher, algorithm chaining and authenticated encryption. The cpa_cy_key.h file contains the API for key generation. The file cpa_cy_rand.h contains the API for random number generation.

Table 15. Cryptographic Symmetric Interface Summary (cpa_cy_sym.h)

Method	Description
cpaCySymInitSession	Initialize a symmetric cryptographic session.
cpaCySymRemoveSession	Remove a symmetric cryptographic session.
cpaCySymPerformOp	Perform a symmetric cryptographic operation.
cpaCySymSessionCtxGetSize	Get the size of the memory the client must allocate in order to store the session context.
cpaCySymQueryStats	Query statistics for symmetric cryptographic operations.

Table 16. Cryptographic Symmetric Key Interface Summary (cpa_cy_key.h)

Method	Description
cpaCyKeyGenSsl	Acceleration of SSL Key Generation.
cpaCyKeyGenTls	Acceleration of TLS Key Generation.
cpaCyKeyGenMgf	Acceleration of mask generation operations.
cpaCyKeyGenQueryStats	Query statistics for symmetric key expansion/generation.

Table 17. Cryptographic Asymmetric Rand Interface Summary (cpa_cy_rand.h)

Method	Description
cpaCyRandGen	Generate random bits or a random number.
cpaCyRandSeed	Seed or perform a seed update on the random data generator. Entropy testing and reseeding are performed automatically on the EP80579, therefore this API returns CPA_STATUS_SUCCESS.
cpaCyRandQueryStats	Query statistics for random bits/number generation.

- **Asymmetric**: The PKE API is contained in various files. The cpa_cy_rsa.h file deals with RSA. The cpa_cy_dsa.h file deals with DSA. The cpa_cy_dh.h file deals with Diffie-Hellman. The cpa_cy_prime.h file deals with prime number generation and testing. The file cpa_cy_ln.h contains the API for large numbers.

Table 18. Cryptographic Asymmetric RSA Interface Summary (cpa_cy_rsa.h)

Method	Description
cpaCyRsaKeyGen	Generate RSA keys.
cpaCyRsaEncrypt	Encrypt a message using an RSA public key.
cpaCyRsaDecrypt	Perform an RSA decrypt or sign operation on the input data.
cpaCyRsaQueryStats	Query statistics RSA operations.

Table 19. Cryptographic Asymmetric Diffie-Hellman Interface Summary (cpa_cy_dh.h)

Method	Description
cpaCyDhKeyGenPhase1	Accelerate Diffie-Hellman phase 1 operations.
cpaCyDhKeyGenPhase2Secret	Accelerate Diffie-Hellman phase 2 operations.
cpaCyDhQueryStats	Query statistics for Diffie-Hellman operations.

Table 20. Cryptographic Asymmetric Large Numbers Interface Summary (cpa_cy_ln.h)

Method	Description
cpaCyLnModExp	Accelerate modular exponentiation. It calculates: result = (base ^ exponent) mod modulus.
cpaCyLnModInv	Accelerate modular Inversion. It calculates: result = (1/A) mod B.
cpaCyLnStatsQuery	Query statistics for Large Number operations.

Table 21. Cryptographic Asymmetric Prime Interface Summary (cpa_cy_prime.h)

Method	Description
cpaCyPrimeTest	Test probabilistically if a number is a prime number.
cpaCyPrimeQueryStats	Query statistics for prime number operations.

12.2.1 Modes of Operation

The Cryptographic API may be called in asynchronous or synchronous modes.

Note: Asynchronous mode is preferred for optimal performance.

12.2.1.1 Asynchronous Operation

In asynchronous mode, the user supplies a callback function to the API. Control returns to the client after the message has been sent to the SSU and the callback gets invoked when the SSU completes the operation. There is no blocking. This mode is preferred for optimal performance.

12.2.1.2 Synchronous Operation

In synchronous mode, the client supplies no callback function pointer (NULL) and the point of execution is held by a semaphore wait internally after a message is successfully passed to the SSU. Upon the completion of the operation, an internal callback function posts to the waiting semaphore and execution will resume. Synchronous mode is therefore blocking and should not be used when invoking the function from a context in which sleeping is not allowed, for example, in interrupt context on Linux. To achieve optimal performance from the API, asynchronous mode is preferred.

12.2.2 Interrupt Operation

The functions in the Cryptographic API may be invoked in both asynchronous mode and synchronous mode. In either case, when the response data is available from the SSU, hardware will inform the system via an interrupt. The QAT-AL component will receive the interrupt and inform the Cryptographic API Library.

Response processing will be performed in a standard OS bottom half (for Linux, this will be a tasklet); this will defer much of the work load into a kernel managed bottom half mechanism without locking the system up by holding the interrupt.

In asynchronous mode, the callback function will be invoked in the context of a tasklet. In synchronous mode, the work queue is de-queued and the client process resumes.

12.2.2.1 Interrupt Coalescing

Interrupt coalescing is the combining of several interrupts into one interrupt. This feature is available on Intel® EP80579 Integrated Processor. The user may configure the duration of time over which to collect the interrupts. See Table 7, "Resource Variables" on page 33 for descriptions of the relevant variables:

- ET_RING_LOOKASIDE_INTERRUPT_COALESCING_ENABLE
- ET_RING_LOOKASIDE_COALESCE_TIMER_NS

12.2.3 Engine and Priority Support

The Cryptographic API is designed to support multiple security services units, or "engines". The engine is specified using the CpaInstanceHandle handle type. This handle will represent a specific engine within the system and will be passed into the symmetric and asymmetric API.

Note: EP80579 with QuickAssist only supports one engine; this is specified through passing in the value CPA_INSTANCE_HANDLE_SINGLE to the API for the acceleration handle.

The API tracks the number of outstanding sessions per handle and for public key, the number of outstanding requests.

The API also has support for priorities per request. In the current release, two levels of priorities are supported: High priority or Normal priority.

The software uses a weighted round robin-based priority scheme. Each ring has an assigned negative 'weight' parameter. Every time a particular ring is polled, its current weight is increased. Once the weight becomes zero, the weight is reloaded with its assigned initial value and the poll moves to the next non-empty ring.

High priority rings have a larger negative value, meaning they are checked more frequently before jumping to the next ring.

12.2.4 Statistics

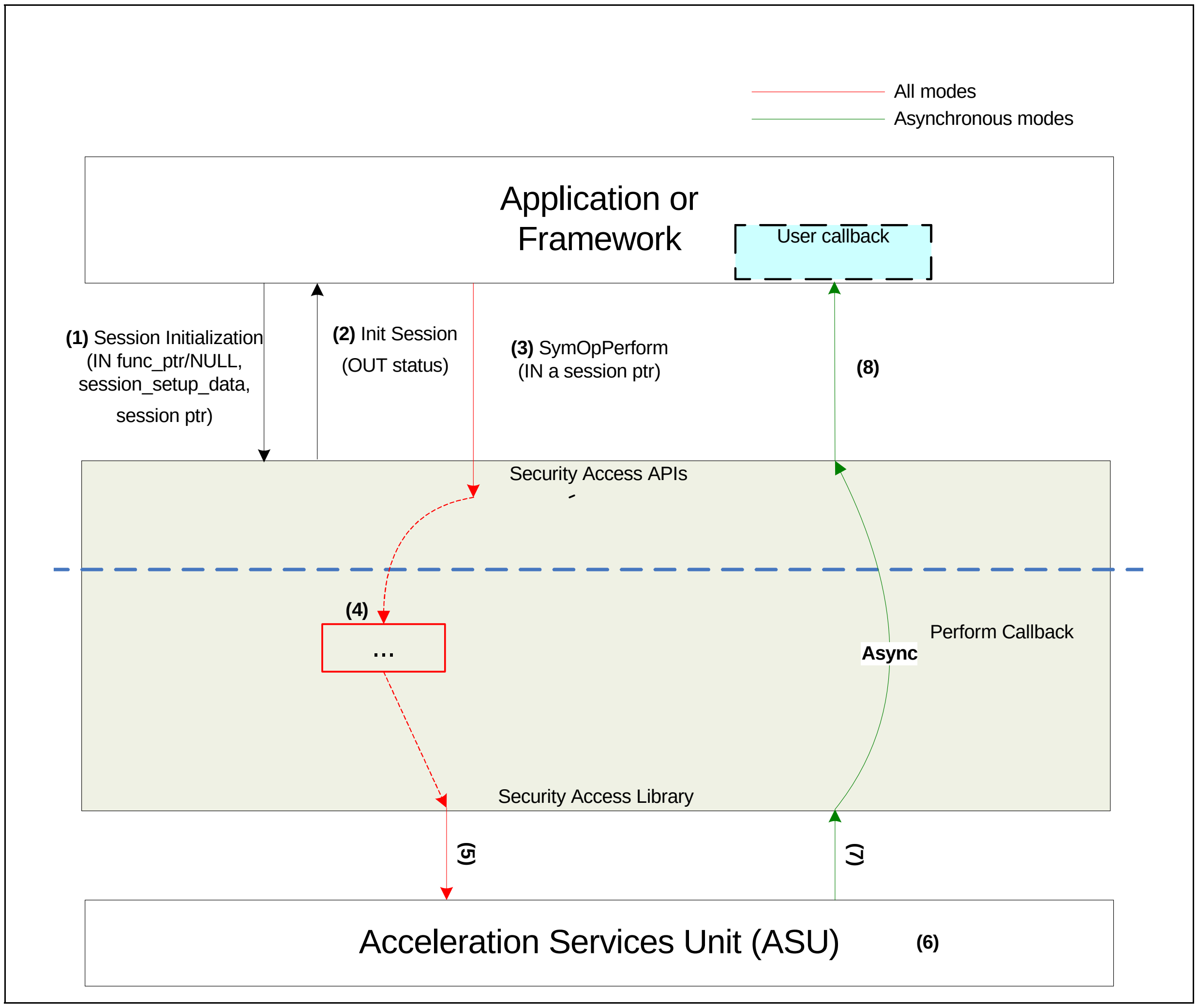
The Cryptographic API supports statistics retrieval and display for the individual symmetric and asymmetric components. For further information, refer to the API manual. The following functions are supported:

- Retrieve statistics (symmetric and asymmetric components). The statistics are retrieved on a per-instance basis - the information stored includes the following:
  - — number of symmetric sessions initialized, removed and errors
  - — numbers of requested operations, completed operations, and failed operations (symmetric and asymmetric components)

12.3 Symmetric Cryptographic API Data Flow

Data flow for the Cryptographic API is shown in Figure 18 and Figure 19 (asynchronous and synchronous). The assumption here is that the hardware is already initialized and ready to process commands.

Figure 18. Symmetric Asynchronous Intel® QuickAssist Technology Cryptographic API Data Flow

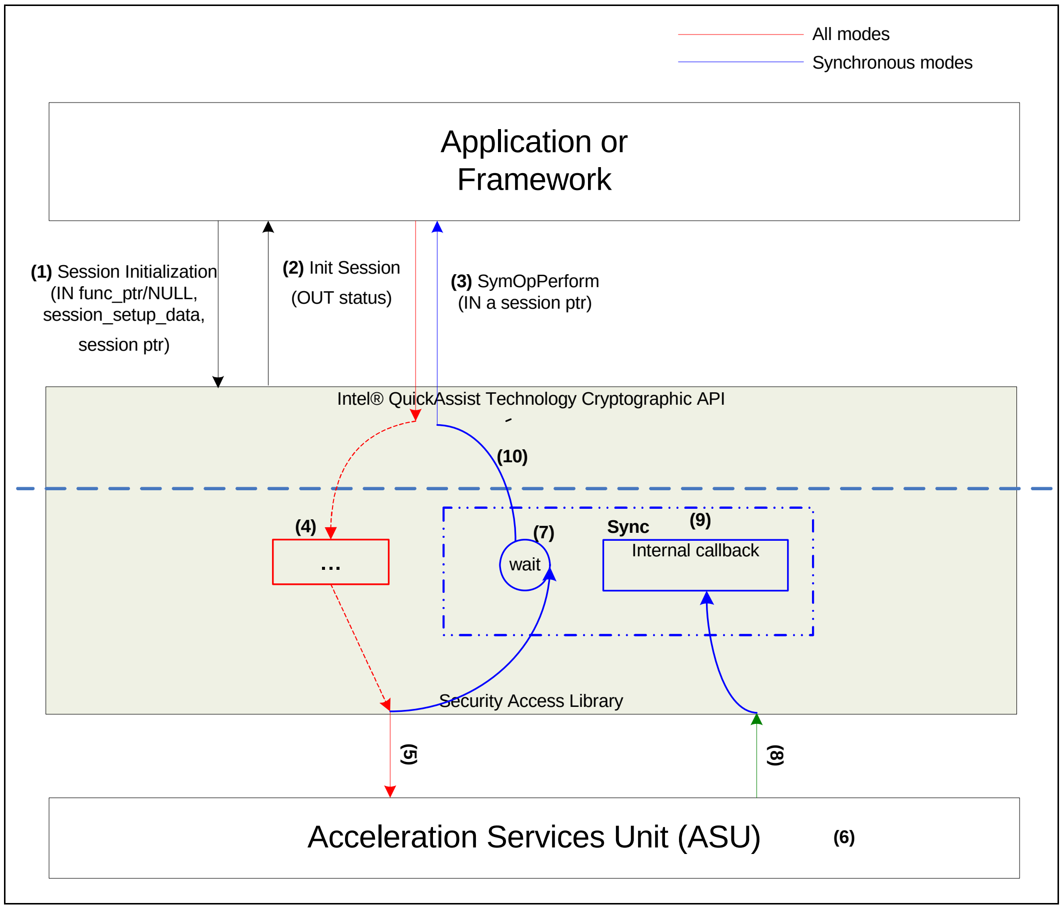

All modes
Asynchronous modes
Application or Framework
User callback
(1) Session Initialization (IN func_ptr/NULL, session_setup_data, session ptr)
(2) Init Session (OUT status)
(3) SymOpPerform (IN a session ptr)
(8)
Security Access APIs
(4)
...
Async
Perform Callback
Security Access Library
(5)
(7)
Acceleration Services Unit (ASU)
(6)

1. The application or framework initializes a session using the API session initialization function, this is a synchronous operation. During session initialization pre-computes for Hash operations will be computed. This will involve an asynchronous call to the Security Services Unit (SSU) which will return the pre-computes. This only occurs during Hash operations. To the external Cryptographic API, this call is synchronous as the asynchronous nature is handled internally in Cryptographic API. The asynchronous part is hidden from the user by means of a queue.
2. Once the session has been initialized, the status of the session initialization is returned to the application or framework, along with the session context handle. The callback parameter is set to the client callback function for asynchronous mode.
3. The application or framework calls the SymOpPerform function to perform a Crypto Operation along with the data pointer on which to perform the operation.

4. The Cryptographic API makes a call to an internal function which handles the operation and understands the message format to send to the SSU.
5. The Cryptographic API functions internally format the data as required by the hardware and send it to the SSU.
6. The SSU performs the required crypto operation.

   Note: At this point, behavior diverges depending on whether the function was invoked synchronously or asynchronously.
7. The SSU after the operation is complete informs the software of the results via an interrupt.
8. The Cryptographic API calls the opPerform completion callback, along with the output data after formatting the data. This call executes in the context of a bottom half.

Figure 19. Symmetric Synchronous Intel® QuickAssist Technology Cryptographic API Data Flow


All modes
Synchronous modes
Application or Framework
(1) Session Initialization (IN func_ptr/NULL, session_setup_data, session ptr)
(2) Init Session (OUT status)
(3) SymOpPerform (IN a session ptr)
Intel® QuickAssist Technology Cryptographic API
(10)
(4)
…
(7)
wait
Sync
(9)
Internal callback
Security Access Library
(5)
(8)
Acceleration Services Unit (ASU)
(6)

1. The application or framework initializes a session using the API session initialization function, this is a synchronous operation. During session initialization pre-computes for Hash operations will be computed. This will involve a asynchronous call to the Security Services Unit (SSU) which will return the pre-computes. This only occurs during Hash operations. To the external Cryptographic API, this call is synchronous as the asynchronous nature is handled internally in Cryptographic API. The asynchronous part is hidden from the user by means of a queue.

2. Once the session has been initialized, the status of the session initialization is returned to the application or framework, along with the session context handle. The callback parameter is set to NULL for synchronous mode.
3. The application or framework calls the SymOpPerform function to perform a Crypto Operation along with the data pointer on which to perform the operation.
4. The Cryptographic API makes a call to an internal function which handles the operation and understands the message format to send to the SSU.
5. The Cryptographic API functions internally format the data as required by the hardware and send it to the SSU.
6. The SSU performs the required crypto operation.

   Note: At this point, behavior diverges depending on whether the function was invoked synchronously or asynchronously.
7. The client process is blocked via a semaphore, see Figure 19.
8. The SSU after the operation is complete informs the software of the results via an interrupt.
9. The Cryptographic API calls the opPerform completion callback, along with the output data after formatting the data. This call executes in the context of a bottom half.
10. An internal callback function is used instead of the client's callback. The point of execution is placed on a work queue after the message is sent to the SSU.
11. After the SSU completes the operation the internal callback is used to de-queue the process on the work queue.
12. The process is de-queued and returns control to the client code.

12.4 Data Format

All input data to the Cryptographic API is in one of the following formats:

- **Flat Buffer:** a simple, unchained buffer of physically contiguous memory
- **Buffer List:** a scatter gather buffer list structure. It is expected that this buffer structure will be used where more than one flat buffer can be provided on an particular API

The supported format(s) vary across the API - see the API manual for details on the formats that are supported by each individual API function.

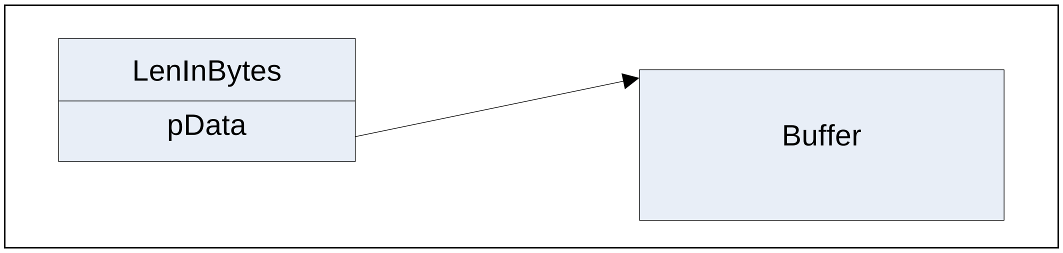
12.4.1 Flat Buffers

Flat buffers are represented by the type CpaFlatBuffer, defined in cpa.h. Figure 20 shows the layout of the flat buffer. The data pointer, pData, points to the start address of the data or payload, stored in Buffer. The length of this buffer specified in bytes is stored in LenInBytes.

The data pointer, pData, is a virtual address, however the actual data pointed to is required to be in contiguous physical memory. This buffer handle is typically used when simple, unchained buffers are needed.

Sample routines for handling flat buffers are provided in the OCF shim code, see [OCF_CODE] for details.

Figure 20. Flat Buffer Diagram



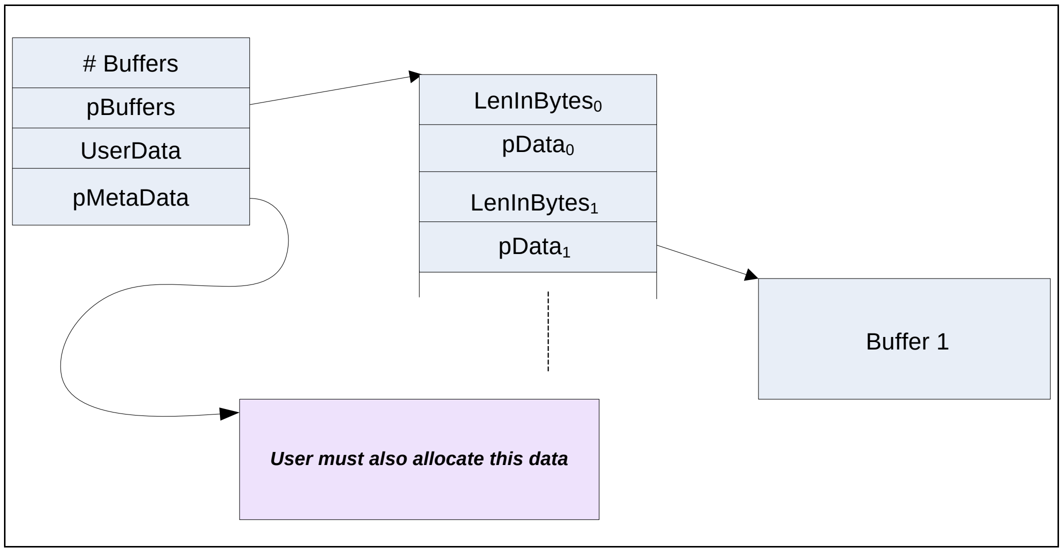
12.4.2 Buffer List

The Cryptographic API uses a scatter gather buffer list structure. This buffer structure is typically used where more than one flat buffer can be provided on an particular API. A Buffer List is defined by the type CpaBufferList, defined in cpa.h. Figure 21 is a graphical representation of a buffer list. The buffer list contains four parameters: the number of buffers in the list, a pointer to an unbounded array of flat buffers, user data and a pointer to meta data required by the API, pMetaData.

The user data is an opaque field and is not read or modified internally by the API. The meta data is required for internal use by the API and the memory for this buffer needs to be allocated by the client as contiguous data. The size of this meta data buffer is obtained by calling a BufferListGetMetaSize function.

Sample routines for converting between Linux sk_buff structures and CpaBufferList are provided in the OCF shim code, see [OCF_CODE] for details.

Figure 21. Buffer List Diagram



12.5 Memory Management

Certain per packet data, such as the Initialization Vector (IV), may be copied depending on whether it is 8-byte aligned or not. This will have an impact on performance. This puts responsibility on the user of the API to be aware of the optimal data alignment for the API they are using. Also the user needs to make sure that the data is available in memory while the Security Services Unit is using it (that is until the callback is invoked).

Note: For details on the most optimal usage of the Cryptographic API, refer to the API manual for the specific interface in question.

The Cryptographic API takes two formats of buffer; CpaFlatBuffer and CpaBufferList. These data types are discussed in sections 12.4.1 and 12.4.2 respectively. For buffer lists it is assumed that each individual buffer in the list is entirely contiguous in memory; that is, the Cryptographic API Library does not perform any scatter-gather operations when forwarding data buffers to the Security Services Unit.

Note: It is **not** required that each buffer in the buffer list is contiguous in memory to the other buffers in a buffer list.

All input and output data buffers will be allocated and freed by the client.

12.6 Endianness and Alignment

All packet data shall be in network byte order (big-endian format) and the Lookaside Security module shall not be required to do endian swaps on the data. PKE buffers will also be passed in big-endian format as per the relevant standards.

Note: There is no endianness associated with randomly generated data.

For optimal performance, data pointers should be 8-byte aligned. In some cases this is a requirement, while in most other cases, it is a recommendation for performance. Please refer to the API manual for optimal usage of the Cryptographic API.

12.7 High-Level API Flow

The following subsections describe the main usage scenarios for the Cryptographic API.

12.7.1 Cryptographic API Initialization and Shutdown

Note: The naming convention used for Initialization/Shutdown APIs is specific to the ASD component rather than the Cryptographic API. These APIs are only called by the ASD component to ensure correct initialization and shutdown procedures.

12.7.1.1 Initialization

On successful completion of this function, Cryptographic API is ready to be started. This sequence of initialization/start/stop/shutdown is controlled by the ASD component to ensure the overall system is correctly configured. The API will be invoked by ASD and is defined as:

```
icp_AsdCfgLacInit()
```

12.7.1.2 Start

This API must be called before the Cryptographic API module will respond to any cryptographic requests, this includes registration of sessions or performing operations. The API will be invoked by ASD and is defined as:

```
icp_AsdCfgLacStart()
```

12.7.1.3 Stop

Stop must be called prior to execution of the shutdown command. Once the Cryptographic API module is shutdown, it will no longer respond to session creation of perform operation requests. It will process all requests/responses still in flight. The API is invoked by the ASD module:

```
icp_AsdCfgLacStop()
```

12.7.1.4 Shutdown

Shutdown is performed only in accordance with the following:

a. If there is an outstanding session, shutdown will be aborted.

b. If Stop has not been invoked and returned successfully, shutdown will be aborted.

c. Otherwise shutdown will complete.

It is the responsibility of ASD to invoke the shutdown command on the Cryptographic API module.

```
icp_AsdCfgLacShutdown()
```

12.8 Intel® QuickAssist Technology Cryptographic API Data Flow

The following sections detail the basic steps involved in performing operations using the Cryptographic API. Sample code and procedures are supplied for a subset of operations, see [SAMPLE_CODE] for details.

12.8.1 Completion of an Operation

1. Asynchronous: A callback will be invoked upon completion of the operation
2. Synchronous: The operation perform function will return

Note: Freeing Memory: For Asychronous mode, user allocated memory may be freed once the callback is called. For Sychronous mode, user allocated memory may be freed once the Operation Perform API returns.

12.8.2 Symmetric Operations

A symmetric operation typically has the following cycle: session initialization, perform one or more operations and session removal when finished.

12.8.2.1 Session Initialization

1. Define a symmetric callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the session
   a. Session setup data
   b. Session context
   — Call session context get size API to get the size

Note: The session context memory must be available to the API for the duration of the session. Other session memory may be freed once the session is initialized.

3. Populate the symmetric session setup data structure
    a. Session priority (normal or high)
    b. Symmetric operation (Cipher, Hash, Auth-Cipher, chained)
    c. Operation setup data structure (Cipher and/or Hash)
    d. Algorithm chaining order
4. Populate the operation setup data structure
- Cipher and/or Hash
- Refer to the API manual for full parameter details
5. Populate the symmetric session setup structure
6. Call the symmetric session initialize API

Now the session is initialized it can be used to perform symmetric operations.

12.8.2.2 Session Removal

When the session is no longer required it may be removed by calling the session removal API.

After the session has been removed the memory allocated for the session context may be freed.

12.8.2.3 Cipher, Hash, Nested and Authentication (Full Packet)

Sample code is provided for Cipher and Hash operations, see [SAMPLE_CODE] and the API manual. The basic steps involved in performing an operation are detailed below. A symmetric operation requires a session for that operation type to be initialized before performing an operation.

1. Initialize a session, see Section 12.8.2.1
2. Allocate memory for the source and destination buffer lists
- For an in-place operation only one buffer list needs to be allocated
3. Allocate memory for the symmetric operation data

Cipher Only:

- Allocate memory for the Initialization Vector (IV)
    — 8-byte aligned for optimal performance
4. Populate the appropriate symmetric operation data structure, see the API manual
5. Call the symmetric operation perform API one or more times
6. Completion of operation, see Section 12.8.1
7. Remove session, see Section 12.8.2.2

12.8.2.4 Partial Packet Variation (Cipher, Hash, Authentication)

The following partial packet variation applies to the full packet sequences described for Cipher, Hash and Authentication (Section 12.8.2.3). Partial packets may be used in a situation where a large packet was segmented on the network.

Change the following steps from full packet requests:

1. Populate the symmetric operation data structure
2. Call the symmetric operation perform API

With the following steps which are used for partial packets:

1. Populate the symmetric operation data structure and set packet type to partial
2. Call the symmetric operation perform API
3. Repeat steps 1 & 2 above for each partial packet which needs to be processed
4. For the final partial packet initialize the symmetric operation data structure and set the packet type to last partial
5. Perform the operation

Note: Steps 1) and 2) may be repeated according to number of partial packets requiring Cipher/Hash/Authentication.

Note: The size of the data to be Hashed or Ciphered must be a multiple of the block size of the algorithm for all partial packets except the last partial.

Note: For Hash/Authentication the digest pointer and the digest verify flag are only used for the last partial.

12.8.2.5 Algorithm Chaining and Authenticated-Encryption

Algorithm chaining involves performing a cipher followed by a hash or a hash followed by a cipher in one operation. Authenticated-encryption involves performing a authenticate followed by a cipher in one operation.

Sample code is provided for algorithm chaining operations; see [SAMPLE_CODE] and the API manual. The basic steps involved in performing an operation are detailed below.

1. Initialize a session, see Section 12.8.2.1
2. Allocate memory for the source and destination buffer lists
   - For an in-place operation, only memory for the source buffer list needs to be allocated
3. Allocate memory for the symmetric operation data
4. Populate the appropriate symmetric operation data structures, see the API manual
5. Call the symmetric operation perform API one or more times
6. Completion of operation, see Section 12.8.1
7. Remove session, see Section 12.8.2.2

12.8.2.6 SSL, TLS Key and MGF Mask Generation

Refer to the API manual for full details of Key and Mask Generation operations.

1. Define a Flat Buffer callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the appropriate operation data structure, see the API manual
   - Fill in the Flat Buffers; pointer to data and length
   - Fill in the options for the operation required
4. Call the appropriate Key or Mask Generation API

5. Completion of the operation, see Section 12.8.1

Note: The API for TLS key operations is based on the TLS 1.1 standard (RFC 4346). Backward compatibility is supported with the legacy TLS 1.0 standard (RFC 2246). The user-defined label should be used for backward compatibility with the client write key, server write key, and iv block. See the Cryptographic API for details of populating CpaCyKeyGenTlsOpData, the operation data structure. Below are some examples of the parameter mapping to the Cryptographic API.

12.8.2.6.1 Setting CpaCyKeyGenTlsOpData Structure Fields

In RFC 4346, Section 6.3 ‘key_block’ is described as:

```
 key_block = PRF(SecurityParameters.master_secret,
                 "key expansion",
                 SecurityParameters.server_random +
                 SecurityParameters.client_random);
```

This maps to the Cryptographic API's CpaCyKeyGenTlsOpData as follows:

```
TLS Key-Material Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_KEY_MATERIAL_DERIVE
    secret = master secret key
    seed = server_random + client_random
    userLabel = NULL
```

12.8.2.6.2 Setting CpaCyKeyGenTlsOpData Structure Fields for backward compatibility

1. In RFC 2246, Section 6.3 ‘final_client_write_key’ is described as:

```
final_client_write_key  = PRF(client_write_key,
                              "client write key",
                              client_random +
                              server_random)[0..15]
```

This maps to the Cryptographic API’s CpaCyKeyGenTlsOpData as follows:

```
TLS User Defined Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED
    secret = client_write_key
    seed = client_random + server_random
    userLabel = "client write key"
```

2. In RFC 2246, Section 6.3 ‘final_server_write_key’ is described as:

```
final_client_write_key  = PRF(server_write_key,
                              "server write key",
                              client_random +
                                    server_random)[0..15]
```

This maps to the Cryptographic API's CpaCyKeyGenTlsOpData as follows:

```
TLS User Defined Derivation:

    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED

    secret = server_write_key

    seed = client_random + server_random

    userLabel = "server write key"
```

3. In RFC 2246, Section 6.3 'iv_block' is described as:

```
 iv_block                = PRF("", "IV block", client_random +

                                    server_random)[0..15]
```

This maps to the Cryptographic API's CpaCyKeyGenTlsOpData as follows:

```
TLS User Defined Derivation:

    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED

    secret = NULL

    seed = client_random + server_random

    userLabel = "IV block"
```

Note: Memory for the user label must be physically contiguous memory allocated by the user. This memory must be available to the API for the duration of the operation, see Section 12.5 for details.

12.8.2.7 Generate Random Data

1. Define a random data callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the Random operation data structure, see the API manual
4. Call the Random data generation perform operation API
5. Completion of the operation, see Section 12.8.1

12.8.3 Asymmetric Operations

Asymmetric operations generally have the following cycle: allocate memory, populate operation structure, perform operation and completion of the operation.

12.8.3.1 Test Prime Number

Sample code is provided for Prime-Test operation see [SAMPLE_CODE].

1. Define a Prime-Test callback function as per the API prototype, see the API manual . If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation

- Inputs: operation data structure

   a. Prime Candidate
   b. Perform GCD test
   c. Perform Fermat test
   d. Number of Miller-Rabin rounds
   e. Perform Lucas test
- Output: Test Passed

3. Populate data for the Prime-Test operation data structure, see the API manual
- Fill in the Flat Buffers; pointer to data and length

4. Call the operation perform Prime Test API
5. Completion of the operation, see Section 12.8.1

12.8.3.2 Diffie-Hellman Phase 1 Key and Phase 2 Private Key Generation

Sample code is provided for Diffie-Hellman Phase operations, see [SAMPLE_CODE].

1. Define a Diffie-Hellman callback function as per the API prototype, see the API manual . If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the appropriate Diffie-Hellman operation data structure, see the API manual
- Fill in the Flat Buffers; pointer to data and length

4. Call the operation perform Phase 1 key generation API
5. Completion of the operation, see Section 12.8.1

12.8.3.3 DSA P, G, Y Parameter Generate

1. Define a DSA callback function as per DSA generic callback API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the appropriate operation data structure, see the API manual
- Fill in the Flat Buffers; pointer to data and length

4. Call the appropriate operation perform DSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.4 DSA R, S, R & S Signature Generation

1. Define a DSA callback function as per DSA generic callback API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the appropriate operation data structure, see the API manual
- Fill in the Flat Buffers; pointer to data and length

4. Call the appropriate operation perform DSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.5 DSA Signature Verification

1. Define a DSA callback function as per DSA generic callback API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate data for the appropriate operation data structure, see the API manual
   - Fill in the Flat Buffers; pointer to data and length
4. Call the appropriate operation perform DSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.6 RSA Key Generation Type 1 and Type 2

1. Define a RSA callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate the RSA operation data structure, see the API manual
   - Fill in the Flat Buffers; pointer to data and length
4. Call the operation perform RSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.7 RSA Encryption and Signature Verification

1. Define a RSA callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate the appropriate RSA operation data structure, see the API manual
   - Fill in the Flat Buffers; pointer to data and length
4. Call the operation perform RSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.8 RSA Decryption and Signature Generation

1. Define a RSA callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation
3. Populate the appropriate RSA operation data structure, see the API manual
   - Fill in the Flat Buffers; pointer to data and length
4. Call the operation perform RSA API
5. Completion of the operation, see Section 12.8.1

12.8.3.9 Large Number Operations - Modular Exponentiation & Inversion

1. Define a Large Number callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.

2. Allocate memory for the operation
3. Populate the appropriate Large Number operation data structure, see the API manual

- Fill in the Flat Buffers; pointer to data and length

4. Call the Large Number operation perform API
5. Completion of the operation, see Section 12.8.1

12.9 Using a Cryptographic Framework

Note: The EP80579 security software release package version 1.0.3 does not support OpenBSD/FreeBSD Cryptographic Framework (OCF), OCF-Linux, or any open source projects such as Openswan*, OpenSSL*, or Racoon*. If your application requires OCF, you must use security software package version 1.0.2 which includes shim software to enable OCF support.

A number of cryptographic frameworks exist within the industry and/or the open source community. These frameworks typically provide software implementations of various cryptographic operations, and allow vendors of cryptographic accelerators to “plug in” their hardware-based implementation underneath. One such cryptographic framework is the OpenBSD/FreeBSD Cryptographic Framework (OCF). OCF is a service virtualization layer that facilitates asynchronous access to cryptographic hardware accelerators. OCF-Linux is a port of this framework to Linux.

A driver has been created which enables the Lookaside Cryptographic features to be accessed via OCF. See the [GET_STARTED_GD] for your operating system for more detailed information.

For customers who already program to the OCF API, this “shim” offers a simple way to utilize the Cryptographic API without changing application code. Programming against a portable API such as OCF protects your software investment, allowing your application to run on any processor supported by OCF, while still taking advantage of the lookaside cryptographic acceleration services when running on Intel® EP80579 Integrated Processor or future silicon supporting the Intel® QuickAssist Technology.

Further information on OCF-Linux can be found here: http://ocf-linux.sourceforge.net

12.10 Accelerating Cryptographic Protocols

Note: The EP80579 security software release package version 1.0.3 does not support OpenBSD/FreeBSD Cryptographic Framework (OCF), OCF-Linux, or any open source projects such as Openswan*, OpenSSL*, or Racoon*. If your application requires OCF, you must use security software package version 1.0.2 which includes shim software to enable OCF support.

Cryptographic protocols, such as IPSec/IKE or SSL, can consume significant computing cycles executing cryptographic operations such as:

- encryption/decryption to ensure confidentiality
- message digests for authentication
- modular exponentiation for key exchange via public key cryptography

These operations can be very compute-intensive, so accelerating these by off-loading the processing from the main processor core, can allow higher throughput or free up cycles for other, higher-value applications.

There are several open-source projects which implement these protocols. These include Openswan* (which implements IPSec), OpenSSL* (which implements SSL/TLS), and Racoon* (which implements IKE on FreeBSD). These in turn can be configured to use OCF for their cryptographic operations. Using the OCF shim described in Section 12.9, customers may be able to take advantage of cryptographic acceleration in the Openswan and OpenSSL software suites.

12.11 Error Handling

In the case of an error occurring when an API function is invoked, the API function will return with the error condition and no callback function will be called. Table 22 shows the status values that are defined in cpa.h, along with their descriptions.

Table 22. Cryptographic API Status Values

Value	Description
CPA_STATUS_SUCCESS	Operation was successful
CPA_STATUS_FAIL	General or unspecified error occurred
CPA_STATUS_RETRY	Recoverable error occurred
CPA_STATUS_RESOURCE	Required resource unavailable
CPA_STATUS_INVALID_PARAM	Invalid parameter supplied
CPA_STATUS_FATAL	Fatal error has occurred

§ §

Appendix A NPF Copyright Notice

The following copyright notice is included because some of the content in this manual (specifically, Section 10.0, “Programming Model” on page 43) references the [NPF API].

Figure 22. NPF Copyright Notice

Note that the Network Processor Forum (NPF) merged with the Optical Internetworking Forum (OIF) in 2006.

§ §